THE UNIVERSITY
of EDINBURGH

Parallel computation techniques for virtual acoustics and physical modelling synthesis

Craig J. Webb

Doctor of Philosophy

Acoustics and Audio Group

University of Edinburgh

2014

Abstract

The numerical simulation of large-scale virtual acoustics and physical modelling synthesis is a computationally expensive process. Time stepping methods, such as finite difference time domain, can be used to simulate wave behaviour in models of three-dimensional room acoustics and virtual instruments. In the absence of any form of simplifying assumptions, and at high audio sample rates, this can lead to simulations that require many hours of computation on a standard Central Processing Unit (CPU). In recent years the video game industry has driven the development of Graphics Processing Units (GPUs) that are now capable of multi-teraflop performance using highly parallel architectures. Whilst these devices are primarily designed for graphics calculations, they can also be used for general purpose computing. This thesis explores the use of such hardware to accelerate simulations of three-dimensional acoustic wave propagation, and embedded systems that create physical models for the synthesis of sound.

Test case simulations of virtual acoustics are used to compare the performance of workstation CPUs to that of Nvidia's Tesla GPU hardware. Using representative multi-core CPU benchmarks, such simulations can be accelerated in the order of 5X for single precision and 3X for double precision floating-point arithmetic. Optimisation strategies are examined for maximising GPU performance when using single devices, as well as for multiple device codes that can compute simulations using billions of grid points. This allows the simulation of room models of several thousand cubic metres at audio rates such as 44.1kHz, all within a useable time scale. The performance of alternative finite difference schemes is explored, as well as strategies for the efficient implementation of boundary conditions.

Creating physical models of acoustic instruments requires embedded systems that often rely on sparse linear algebra operations. The performance efficiency of various sparse matrix storage formats is detailed in terms of the fundamental operations that are required to compute complex models, with an optimised storage system achieving substantial performance gains over more generalised formats. An integrated instrument model of the timpani drum is used to demonstrate the performance gains that are possible using the optimisation strategies developed through this thesis.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Craig J. Webb*)

Publications

Elements of this thesis have been published in the following papers. The contribution of the current author is detailed in each case.

1. C. Webb and S. Bilbao, "Computing Room Acoustics with CUDA - 3D FDTD schemes with boundary losses and viscosity", In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing* (ICASSP), Prague, Czech, 2011, pp. 317-320.

   The author produced the entire paper, with the exception of Section 2 which details the numerical design of the finite difference scheme.

2. C. Webb and S. Bilbao, "Virtual Room Acoustics: A Comparison of techniques for computing 3D-FDTD schemes using CUDA", In *Proceedings of the 130th Convention of the Audio Engineering Society*, vol. 2, London, UK, 2011, pp. 1163-1169.

   The author produced the entire paper, with the exception of the numerical design of the finite difference scheme as detailed in the previous paper.

3. C. Webb and S. Bilbao, "Binaural Simulations using audio rate FDTD schemes and CUDA", In *Proceedings of the 15th International Conference on Digital Audio Effects*, York, UK, 2012, pp. 97-100.

   The author is responsible for the entire paper, with the exception of the numerical design of the Enquist Majda boundary condition.

4. S. Bilbao and C. Webb, "Timpani Drum Synthesis in 3D on GPGPUs", In *Proceedings of the 15th International Conference on Digital Audio Effects*, York, UK, 2012, pp. 269-276.

   In this paper the numerical design and initial prototype modelling is by Bilbao. The author is responsible for the GPU implementation and optimisation techniques, and the resulting performance data.

5. J. Sheaffer, C. Webb and B. Fazenda, "Modeling Binaural Receivers in Finite Difference Simulation of Room Acoustics", In *Proceedings of Meetings on Acoustics: International Congress on Acoustics*, vol. 19, Montreal, Canada, 2013, p. 15098.

   The theoretical and analysis sections of this paper are by Sheaffer. The author contributed the large-scale GPU testing data on which the results are based, having implemented the computational models.

6. C. Webb and A. Gray, "Large-scale virtual acoustics simulation at audio rates using three-dimensional finite difference time domain and multiple GPUs", In *Proceedings of Meetings on Acoustics: International Congress on Acoustics*, vol .19, Montreal, Canada, 2013, p. 70092.

   The author produced the entire paper, with manuscript editing by Gray.

7. S. Bilbao, B. Hamilton, A. Torin, C. Webb, P. Graham, A. Gray, K. Kavoussanakis and J. Perry, "Large scale physical modeling sound synthesis", In *Proceedings of the Stockholm Music Acoustics Conference*, Stockholm, Sweden, 2013, pp. 593-600.

   For this review paper, the author contributed Section 5 on GPU implementation, and elements of Section 6.4 showing benchmark CPU and GPU performance analysis.

8. C. Webb, "Computing virtual acoustics using the 3D finite difference time domain method and Kepler architecture GPUs" , In *Proceedings of the Stockholm Music Acoustics Conference*, Stockholm, Sweden, 2013, pp. 648-653.

   The author is responsible for the entire paper.

9. B. Hamilton and C. Webb, "Room acoustics modelling using GPU accelerated finite difference and finite volume methods on a face-centered cubic grid", In *Proceedings of the 16th International Conference on Digital Audio Effects*, Maynooth, Ireland, 2013, pp. 336-343.

   For this paper, the author produced GPU implementations of the various finite schemes that are tested, and the performance data used in Section 8.

10. S. Bilbao and C. Webb, "Physical modeling of timpani drums in 3D on GPGPUs", *Journal of the Audio Engineering Society*, vol. 61, no. 10, pp. 737-748, 2013.

    In this paper the numerical design and initial prototype modelling is by Bilbao. The author is responsible for the GPU implementation and optimisation techniques, the resulting performance data, and the abstracted multi-instrument model.

Acknowledgements

This research has been supported by University of Edinburgh scholarships awarded by the College of Humanities and Social Science and the Edinburgh College of Art, and by the European Research Council under Grant StG-2011-279068-NESS.

The NESS project (Next generation sound synthesis) began in the second year of my PhD studies, and I am very fortunate to have been a part of it from the outset. When I began my PhD as the only student working with Stefan I had no idea that I would very soon be part of a large, dedicated, research group.

I would like to thank the many colleagues that I have met along the way, both from the MSc course and the NESS project group, that really made my time at Edinburgh the experience that it was. In particular I have to thank Brian Hamilton, Alberto Torin, and Dr. Michele Ducceschi for their collaboration on various papers and elements of my work.

I would also like to thank my family and Eleni for their continued support throughout my four and half years of MSc and then PhD study. Finally I wish to thank my excellent supervisors Dr. Stefan Bilbao and Dr. Alan Gray, without whom none of this would have been possible.

Table of Contents

Part I

Introduction and background theory

Chapter 1

Introduction

1.1 Introductory remarks

Physical modelling synthesis refers to a group of techniques whose goal is to produce realistic and controllable digital models of musical instruments. Unlike sample-based systems which make use of recordings of actual instruments, these methods compute audio from mathematical principles of musical acoustics. Three-dimensional physical modelling synthesis attempts to create complete virtual models of acoustic instruments which can capture the entire sonic range of their expressive capability. The aim is to compute the full wave behaviour of models embedded in a virtual acoustic environment in order to create high fidelity sound synthesis of real, or indeed hypothetical, acoustic instruments.

One approach to these physics-based simulations is to model wave propagation using time stepping numerical methods, such as finite difference time domain (FDTD). However, at audio sample rates such as 44.1kHz this approach can become extremely computationally expensive. The calculation of two-dimensional systems, for example nonlinear membranes used in drums, and especially three-dimensional wave propagation in large-scale acoustic environments, can lead to simulations that require many hours or even days to execute in a serial manner on a standard workstation computer.

This thesis examines the methods that can be used to accelerate the computation of these simulations using parallel programming techniques on both Central Processing Units (CPUs) and Graphics Processing Units (GPUs). The emphasis here is on using hardware that is generally available for desktop computing, as the overall purpose in sound synthesis is to develop systems which can be used by musicians and composers rather than to run simulations on highly restricted supercomputing hardware.

Recent advances in GPU technology have led to their use as general purpose processors outside of the graphics domain for which they were originally intended. Their highly parallel architecture, containing many hundreds or thousands of cores, offers the prospect of performance gains over standard CPU computation for certain types of algorithms. One of the major benefits of employing FDTD methods to discretise wave equation systems is the high level of data independence available at each time step of a simulation.

1.2 Thesis objectives and outline

The main aim of this thesis is to demonstrate the performance acceleration that is achievable using GPU devices for the specific types of computations required for large-scale three-dimensional physical modelling synthesis using FDTD schemes. It seeks to establish a set of implementation strategies that can be used to produce optimal performance on the GPU, focusing on the use of Nvidia's CUDA platform. This requires the examination of sparse linear algebra techniques, as well as computations using arrays that may contain billions of elements.

In attempting to assess the performance gains that are achievable through the use of GPUs, appropriate *benchmarks* are required for CPU performance. The CPU hardware used throughout this thesis are the Intel i7 and Xeon processors as found in current desktop and workstation machines. These are multi-core processors, and so optimised multi-threaded code is used to assess the performance potential of the hardware and obtain realistic benchmark figures. These are then compared to the performance of CUDA code running on Nvidia Tesla GPUs, using both the Fermi and the most recent Kepler architecture devices.

This thesis is broadly divided into four parts. Part I contains this introduction, along with background theory and relevant literature concerning virtual acoustics and physical modelling, parallel programming on GPUs using CUDA, and a review of basic linear algebra operations. Part II develops techniques for the acceleration of three-dimensional wave equation schemes for virtual acoustic simulations, and consists of three chapters.

Computing solutions to the 3D wave equation

The first of these chapters examines the computation of the most basic FDTD scheme for the three-dimensional wave equation. A test simulation is used to demonstrate the performance potential of multi-threaded CPU code, followed by an examination of optimisation strategies for single GPU device CUDA code. These strategies involve the use of different threading models, the application of shared memory, and cache optimisations. Testing is performed using both single and double precision floating-point arithmetic, as GPU performance varies significantly according to precision level. This is followed by an analysis of multi-device implementations using CUDA that makes use of four GPU cards simultaneously to create large-scale simulations.

Performance of alternative schemes

The second chapter in this part examines alternative finite difference schemes for the three-dimensional wave equation. The staggered grid formation is compared to the basic second order form used in the previous chapter. This is then followed by a comparison of schemes that have different characteristics from the basic scheme, in terms of dispersion and cutoff frequencies. The efficiency of the 27-point interpolated wideband scheme and a 13-point scheme on a face-centered cubic grid is examined.

Virtual acoustic simulations

The final chapter in this part assesses the implementation issues and the impact on efficiency of simple state-free boundary conditions, as well as those which require extra state data to be held in order to implement the boundary condition. It then details large-scale auralizations that use all available memory across four GPU devices, with data grids that contain billions of points. Issues relating to floating-point precision are considered, as well as the inclusion of viscosity into the basic scheme to improve the overall audio quality of the results.

Part III of this thesis concerns the integration of embedded physical models of instruments into the virtual acoustic environments described in Part II. It consists of two chapters.

Basic linear algebra operations

The implementation of complex embedded models often requires the use of linear algebra operations. These operations involve vectors and *sparse* matrices. This chapter examines the performance of basic vector operations, followed by an analysis of matrix by vector multiplication using a variety of sparse matrix storage formats. Nvidia's own CUSPARSE library functions are compared to custom written functions using alternative matrix formats, specifically the DIA and ELLPACK forms. The efficiency of these formats for the particular type of matrices produced by FDTD methods is detailed for both CPU and GPU devices. A comparison of matrix form to 'unrolled' matrix-free formation is also reviewed.

An integrated model of the timpani drum

This chapter brings together the techniques and optimisation strategies developed in this thesis to demonstrate the performance gains achievable for an integrated model of the timpani drum. A detailed analysis is given for the various computational stages of the time step iteration, comparing different implementations of the model. These different implementations use the sparse matrix storage formats examined in Chapter 6, as well a version that makes use of matrix-free operations. The chapter concludes with details of an abstracted instrument system that allows any number of timpani to be embedded inside a virtual room or hall simulation. An example using four timpani drums played simultaneously inside a room model is demonstrated.

Finally, Part IV of the thesis contains a summary of the results, followed by the concluding remarks.

Chapter 2

Background theory and literature

The acceleration of large-scale three-dimensional physical modelling synthesis clearly encompasses a number of different subject areas. This chapter reviews background theory and relevant literature in four main fields: virtual acoustic simulations, physical modelling synthesis of instruments, parallel programming using graphics processing units, and sparse linear algebra.

2.1 Virtual acoustic simulations

Virtual acoustic simulations attempt to create a model of sound propagating in a virtual space. They seek to model the sound sources, the acoustics of the environment, and the listener, in as great a detail as possible [1]. When such models are rendered, the output can be termed an *auralization* [2].

Artificial reverberation can be created using a variety of different techniques, such as analogue or digital delay units [3], or physical items such as spring or plate reverbs (see [4] for a detailed review of reverberation techniques). However, virtual acoustic simulations aim to accurately model the acoustic characteristics, rather than merely giving a generalised impression of reverberation. This process can be applied to areas such as architectural design [5], computer gaming [6], film [7], and virtual reality settings [8]. The modelling techniques typically fall into one of two categories, geometric or wave-based, and make use of general physical principles of acoustics.

2.1.1 Principles of acoustics and the wave equation

Acoustic waves are fluctuations in pressure in a compressible fluid. As waves propagate there are local fluctuations in particle density, and so variations in pressure and particle velocity occur [9]. Under ideal conditions the velocity of propagation is constant, which is based on the assumption that deviations in the acoustic pressure are small compared to the mean value [10]. The behaviour can be described mathematically by two fundamental laws [11], conservation of mass and conservation of momentum, which are expressed as

$$\frac{\partial P}{\partial t} = -\rho c^2 \nabla \cdot \mathbf{v} \qquad\qquad \rho \frac{\partial \mathbf{v}}{\partial t} = -\nabla P \tag{2.1}$$

where $P(x, y, z)$ is the acoustic pressure, $\mathbf{v}(x, y, z)$ is the particle velocity, ρ is the instantaneous density, c is the speed of sound, $\frac{\partial}{\partial t}$ is the partial time derivative, $\nabla\cdot$ is the divergence operator, ∇ is the gradient, and $(x, y, z) \in \mathcal{D} \subset \mathbb{R}^3$. Here $\mathcal{D}$ is the domain of the problem, and one boundary condition is specified at every point on $\partial\mathcal{D}$, the boundary of $\mathcal{D}$. These two equations can be combined to give a single differential equation, the *wave equation*, by eliminating the particle velocity

$$\frac{\partial^2 P}{\partial t^2} \cdot \frac{1}{\rho c^2} = -\nabla \cdot \frac{\partial \mathbf{v}}{\partial t} \tag{2.2}$$

$$\nabla \cdot \frac{\partial \mathbf{v}}{\partial t} = -\frac{1}{\rho} \nabla^2 P \tag{2.3}$$

$$\frac{\partial^2 P}{\partial t^2} = c^2 \nabla^2 P \tag{2.4}$$

where ∇^2 is the three-dimensional Laplacian, given in Cartesian coordinates by

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \tag{2.5}$$

This wave equation can also be expressed as

$$\frac{\partial^2 \Psi}{\partial t^2} = c^2 \nabla^2 \Psi \tag{2.6}$$

where $\Psi(x, y, z, t)$ is a velocity potential given by

$$P = \rho \frac{\partial \Psi}{\partial t} \qquad\qquad \mathbf{v} = -\nabla \Psi \tag{2.7}$$

2.1.2 Geometric methods

Geometric methods for acoustic modelling are long established, and are still used as the basis for commercial software tools such as ODEON and CATT [12]. The core assumption in such methods is that at high frequencies, where the wavelength is much smaller than the room dimensions, sound can be modelled as a "ray" rather than as a wave [13]. By calculating the paths of these rays, a detailed impulse response can be created for the virtual environment. Various methods exist, differing in their approach to computing the pathways.

Ray tracing method

The ray tracing method was first detailed by Krokstad et al. [14], and represents one of the earliest attempts at computer modelling for acoustic simulations. The energy emitted from a source is described by a finite number of rays, whose paths are calculated through the domain and reflect from boundaries. If the rays pass through a defined volume representing a receiver, the intensity of the ray at that point is stored. In that manner an impulse response can be obtained, given sufficient ray paths have been computed.

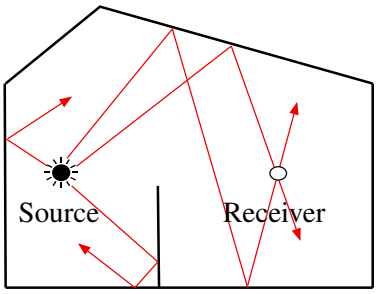


Figure 2.1 *Ray tracing method. Multiple rays are emitted from the source.*

Reflection from boundaries may be either specular, where the angle of incidence is equal to the angle of reflection, or diffuse (reflected in many directions). The production of frequency-dependent reflections can be more computationally expensive.

The accuracy of ray tracing is dependent on the number and direction of the ray paths that are issued from the source, as well as the geometry. This may be a regular distribution around the source, or some other statistical distribution, and many thousands of ray paths may need to be computed to obtain a reasonable level of accuracy. The ray tracing method is also used in graphics rendering applications, and has been implemented for room acoustics using graphics processing units [15].

Image source method

The image source method for acoustic modelling was first implemented in the 1970s [16] [17]. Unlike ray tracing, the paths between the source and receiver are not calculated directly. Instead, the source location is mirrored through the domain boundaries. The distance between the source and these reflected 'images' is used to determine the receiver arrival time. By storing the energy, angle, and time of arrival of each image path an echogram can be created and used as an impulse response.

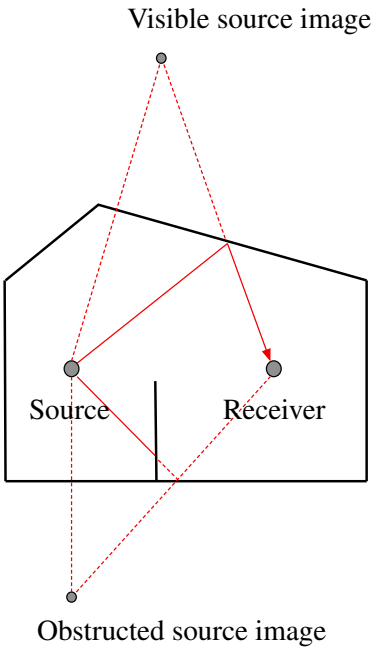


Figure 2.2 *Image source method. The source is mirrored through the boundaries.*

The main benefit of the image source method is that calculated pathways can be identified exactly, and it is particularly useful in calculating early reflections in simple geometries. The method becomes significantly less efficient with an increase in the number of reflections. For domains that are non-rectangular, the process of computing the image sources is far more complex [18]. This may include validating the ray paths to check for visibility and any obstructions, which is performed by a backward reconstruction of the path from the receiver location, via all reflections at the boundaries, to the source.

In terms of boundary losses, specular reflections are easily implemented, and diffuse reflections have been considered. The image source method is often combined with ray tracing to create efficient hybrid geometric models [19].

Beam tracing method

Beam tracing methods are an attempt to minimise the computational cost of performing the image source technique for complex geometries [20]. Specifically, the technique seeks to avoid or reduce the path validity checks required in the image source approach [21]. Rather than using individual rays, beams are formed from the source point and projected at each boundary. The cross-sectional area of these beams can be in the shape of a circle or polygon. When a beam meets a boundary, the intersection is reflected as a beam which may be separated from the transmission beam.

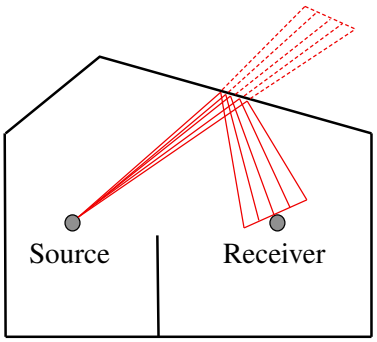


Figure 2.3 *Beam tracing method. Beams, rather than rays, are emitted from the source.*

Beam tracing is typically performed as a two stage process. Firstly there is a pre-computation stage that leads to the determination of a beam-tree, which is a hierarchical data structure that stores information relating to the paths between source and receiver. Secondly there is a rendering stage that uses the image source method to compute the valid reflection paths.

Other geometric techniques

There are various other techniques which exist, still based on a geometric approach. Radiosity rendering is a method used in three-dimensional graphics processing for calculating the diffuse reflection in scenes. By introducing a time dependency to allow for the propagation velocity of sound waves, this approach has been demonstrated for acoustic simulations [22]. The major distinction from other geometric methods is the assumption that all boundary reflections are diffuse, and thus there is no need to compute angles of incidence and reflection. Another more recent approach, the acoustic radiance transfer method, is based on a room acoustics rendering equation for computing energy propagation [23].

2.1.3 Wave-based methods

Although geometric methods have been used in many different applications, their main weakness is the difficulty in modelling low frequency wave behaviour such as the diffraction effect. By contrast, implementations of wave-based methods model full wave behaviour employing some form of numerical approximation to the wave equation with appropriate boundary conditions (as demonstrated in Figure 2.4). Whilst most acoustic modelling methods are used to create an impulse response as their output, wave-based methods computed over time can also be used in a dynamic manner i.e. sources and receivers can move around during the calculation of a simulation [24].

The common feature of wave-based methods is *mesh discretisation*, where the continuous mathematical description of some or all of the domain is replaced by a discrete grid of inter-connecting nodes. The values at these nodal points may represent an acoustic property such as pressure or velocity potential, and are calculated using some computational method.

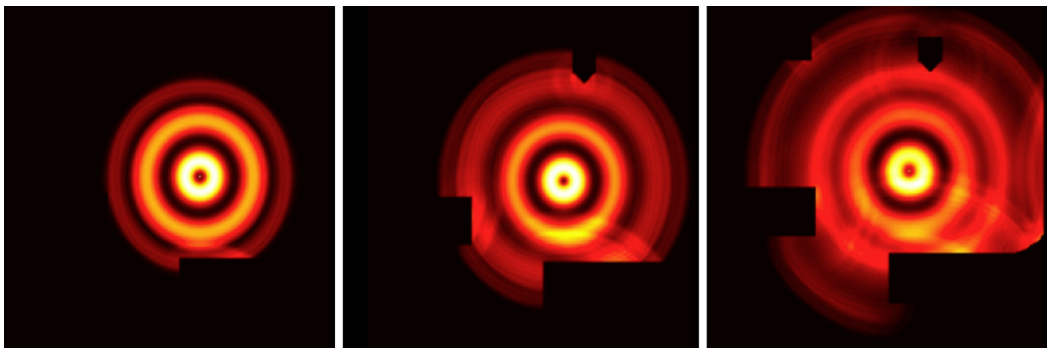


Figure 2.4 *Progressive snapshots of the time evolution of a wave field in a complex three-dimensional environment.*

Again, there are a number of different approaches to wave-based methods, of which finite element (FEM), boundary element (BEM), and the digital waveguide mesh (DWM) are discussed in this section here, and finite difference time domain (FDTD) in section 2.1.4.

Finite element method

The finite element method is a computational technique used to obtain approximate solutions to boundary value problems, and is used extensively in engineering [25]. The method in its present form was introduced by Courant in 1942, which itself was a development of work by Ritz, Galerkin, and others [26]. It has become the dominant method for the numerical solution of problems in areas such as structural and solid mechanics and structural analysis, but is used alongside finite difference methods in areas such as fluid mechanics [27]. The basic components of the method are a mesh which covers the domain, and then for each element the construction of a finite dimensional subspace consisting of piecewise polynomials (basis functions) [28].

The method is generally computed in two stages; a pre-computation stage that deals with the construction of the mesh and the choice of basis functions, followed by a procedure solving a system of linear equations. The finite element method has been used for acoustic modelling, for example in [29], and clearly lends itself to irregular domains, but for mid to high frequency modelling the technique is very computationally expensive.

Boundary element method

The boundary element method makes use of the boundary integral equations. It can be derived from the Kirchhoff-Helmholtz integral theorem, which states that the solution to the Helmholtz equation inside a domain can be found from an integral over the boundary [30]. Using this approach, a discretisation of the boundaries allows the boundary pressure to be computed, which then allows the calculation of pressure at points inside the domain. This can also be applied to energy and intensity variables [31].

Computationally, the system is formulated using matrices that relate each boundary element to every other, including itself, and unlike FEM this results in dense matrices rather than sparse ones. For a detailed comparison of FEM and BEM for acoustic modelling see Kopuz et al. [32].

Digital waveguide mesh

The digital waveguide mesh makes use of multi-dimensional combinations of digital waveguides to discretise the domain (see section 2.2.2 for detail concerning digital

waveguides). These meshes are systems of bidirectional delay lines connected by scattering junctions, as shown in Figure 2.5.

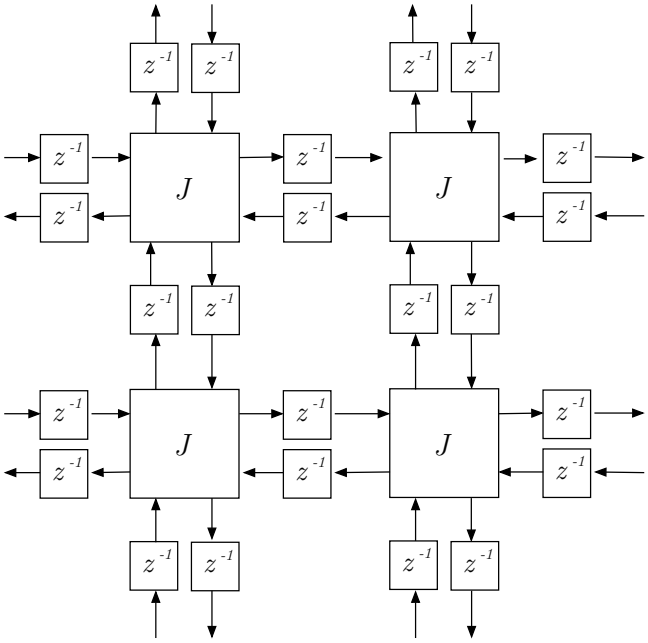


Figure 2.5 *A 2D regular rectangular digital waveguide mesh with scattering junctions J, and delay lines* z^{-1}.

These structures are expanded to cover the entire virtual domain, with some form of boundary termination, in order to model the acoustic wave propagation in the space. At each junction conservation of energy laws are applied, analogous to Kirchhoff's Laws for electrical circuits, and expressed in terms of sound pressure and particle velocity. The sound pressure of the waveguide is given by the sum of waves travelling in all directions. These were first demonstrated for acoustic modelling in two-dimensional schemes by Smith and van Duyne [33], and then in three-dimensional schemes by Savioja [34].

Whilst digital waveguides in one dimension have a significant computational efficiency advantage over other FDTD schemes, this is not the case for two-dimensional and three-dimensional implementations [35]. In these cases the FDTD scheme can compute the same solution at approximately half the computational cost. Digital waveguides are equivalent to FDTD schemes, and can be rewritten as such schemes[36]. However, the DWM approach has been widely researched [37] [38] [39], and full-scale applications have been created using this method [40].

2.1.4 3D finite difference time domain method

The finite difference method is one of the oldest and simplest methods for approximating the solution to partial differential equations, dating back to Courant et al. from 1928 [41] (see [42] for an overview of finite difference techniques). Techniques based on this method are used in many diverse fields, from electromagnetics [43], to seismology [44] and geophysics [45], and of course acoustics [46]. The finite difference time domain method employs finite difference approximations to the spatial and temporal derivatives of the differential equations. It can be viewed as a special case of more general *finite volume* methods [47], when applied to regular meshes [48].

Grids and operators

The solution to the wave equation (2.6) over a rectilinear grid is approximated by a grid function $\Psi^n_{l,m,p}$ representing an approximation to Ψ at $x = lX$, $y = mX$, $z = pX$, and $t = nT$ for integers l,m,n,p. Here, T is the time step and X is the grid spacing (the distance between adjacent spatial grid points). The centred finite difference operators for the second derivatives in the wave equation can be defined as

$$\delta_t^2 \Psi^n_{l,m,p} \equiv \frac{1}{T^2}\left(\Psi^{n+1}_{l,m,p} - 2\Psi^n_{l,m,p} + \Psi^{n-1}_{l,m,p}\right) \cong \frac{\partial^2}{\partial t^2}\Psi \tag{2.8}$$

$$\delta_x^2 \Psi^n_{l,m,p} \equiv \frac{1}{X^2}\left(\Psi^{n}_{l+1,m,p} - 2\Psi^n_{l,m,p} + \Psi^{n}_{l-1,m,p}\right) \cong \frac{\partial^2}{\partial x^2}\Psi \tag{2.9}$$

$$\delta_y^2 \Psi^n_{l,m,p} \equiv \frac{1}{X^2}\left(\Psi^{n}_{l,m+1,p} - 2\Psi^n_{l,m,p} + \Psi^{n}_{l,m-1,p}\right) \cong \frac{\partial^2}{\partial y^2}\Psi \tag{2.10}$$

$$\delta_z^2 \Psi^n_{l,m,p} \equiv \frac{1}{X^2}\left(\Psi^{n}_{l,m,p+1} - 2\Psi^n_{l,m,p} + \Psi^{n}_{l,m,p-1}\right) \cong \frac{\partial^2}{\partial z^2}\Psi \tag{2.11}$$

Regular second order scheme

The most basic finite difference scheme for the wave equation is the scheme that is second order in time. Expanding the Laplacian from equation (2.6) gives

$$\frac{\partial^2 \Psi}{\partial t^2} = c^2\left(\frac{\partial^2 \Psi}{\partial x^2} + \frac{\partial^2 \Psi}{\partial y^2} + \frac{\partial^2 \Psi}{\partial z^2}\right) \tag{2.12}$$

Applying the spatial and temporal difference operators leads to

$$\Psi^{n+1}_{l,m,p} - 2\Psi^{n}_{l,m,p} + \Psi^{n-1}_{l,m,p} = \frac{c^2T^2}{X^2}(\Psi^{n}_{l+1,m,p} + \Psi^{n}_{l-1,m,p} + \Psi^{n}_{l,m+1,p} + \Psi^{n}_{l,m-1,p} + \Psi^{n}_{l,m,p+1} + \Psi^{n}_{l,m,p-1} - 6\Psi^{n}_{l,m,p}) \tag{2.13}$$

Defining the Courant number $\lambda = \frac{cT}{X}$,
and $S = \Psi^{n}_{l+1,m,p} + \Psi^{n}_{l-1,m,p} + \Psi^{n}_{l,m+1,p} + \Psi^{n}_{l,m-1,p} + \Psi^{n}_{l,m,p+1} + \Psi^{n}_{l,m,p-1}$,
leads to the standard "nearest neighbour" scheme

$$\Psi^{n+1}_{l,m,p} = (2 - 6\lambda^2)\Psi^{n}_{l,m,p} + \lambda^2 S - \Psi^{n-1}_{l,m,p} \tag{2.14}$$

Stability

An FDTD scheme is *stable* if the system solutions do not grow exponentially. This can be shown by von Neumann analysis [42], which is an application of Fourier analysis. This involves investigating the behaviour of a test solution of the form

$$\Psi^{n}_{l,m,p} = z^n e^{ih(l\beta_x + m\beta_y + p\beta_z)} \tag{2.15}$$

where $z = e^{j\omega T}$ in radial frequency ω, h is the grid spacing, and β_x, β_y, β_z are the components of the wave number, to determine which values of the Courant number λ the scheme is stable. Substituting into the update scheme (2.14) gives the characteristic equation

$$z - 2 + 4\lambda^2 \left(\sin^2(\beta_x h/2) + \sin^2(\beta_y h/2) + \sin^2(\beta_z h/2)\right) + z^{-1} = 0 \tag{2.16}$$

The solutions in z will lie inside the unit circle when

$$0 \leq \lambda^2 \left(\sin^2(\beta_x h/2) + \sin^2(\beta_y h/2) + \sin^2(\beta_z h/2)\right) \leq 1 \tag{2.17}$$

As the $\sin^2$ function can only have values between 0 and 1, this leads to the standard stability condition

$$\lambda \leq \frac{1}{\sqrt{3}} \tag{2.18}$$

known as the Courant-Friedrichs-Lewy condition [41]. At the Courant limit where $\lambda = 1/\sqrt{3}$, the update scheme (2.14) reduces to

$$\Psi^{n+1}_{l,m,p} = \frac{1}{3}S - \Psi^{n-1}_{l,m,p} \tag{2.19}$$

In audio applications this implies, for a given sample rate, a minimum spacing

$$X \geq \sqrt{3} \cdot cT \tag{2.20}$$

Staggered grid scheme

Rather than the single field finite difference scheme given above, the wave equation can also be discretised into a coupled finite difference scheme in both pressure and velocity. Consider again the equations describing conservation of mass and momentum, given as

$$\frac{\partial P}{\partial t} = -\rho c^2 \nabla \cdot \mathbf{v} \qquad\qquad \rho \frac{\partial \mathbf{v}}{\partial t} = -\nabla P \tag{2.21}$$

Applying finite differences to the derivatives leads to the following update scheme over staggered grids, where v_x, v_y, and v_z are the components of the velocity field

$$(v_x)^{n+\frac{1}{2}}_{l+\frac{1}{2},m,p} = (v_x)^{n-\frac{1}{2}}_{l+\frac{1}{2},m,p} - \frac{T}{\rho X}\left(P^n_{l+1,m,p} - P^n_{l,m,p}\right) \tag{2.22}$$

$$(v_y)^{n+\frac{1}{2}}_{l,m+\frac{1}{2},p} = (v_y)^{n-\frac{1}{2}}_{l,m+\frac{1}{2},p} - \frac{T}{\rho X}\left(P^n_{l,m+1,p} - P^n_{l,m,p}\right)$$

$$(v_z)^{n+\frac{1}{2}}_{l,m,p+\frac{1}{2}} = (v_z)^{n-\frac{1}{2}}_{l,m,p+\frac{1}{2}} - \frac{T}{\rho X}\left(P^n_{l,m,p+1} - P^n_{l,m,p}\right)$$

$$\begin{aligned} P^{n+1}_{l,m,p} = P^n_{l,m,p} - \frac{\rho c^2 T}{X}\Big(&(v_x)^{n+\frac{1}{2}}_{l+\frac{1}{2},m,p} - (v_x)^{n+\frac{1}{2}}_{l-\frac{1}{2},m,p} + (v_y)^{n+\frac{1}{2}}_{l,m+\frac{1}{2},p} \\ &-(v_y)^{n+\frac{1}{2}}_{l,m-\frac{1}{2},p} + (v_z)^{n+\frac{1}{2}}_{l,m,p+\frac{1}{2}} - (v_z)^{n+\frac{1}{2}}_{l,m,p-\frac{1}{2}}\Big) \end{aligned} \tag{2.23}$$

In the staggered grid formation, each pressure node is surrounded by velocity nodes, in three fields, that are positioned half a grid point away. Each velocity node is updated using the two pressure node neighbours in its axis, and then each pressure node is updated using the six velocity neighbours, as shown in Figure 2.6. This scheme is still equivalent to the regular second order formation [49]. See Section 4.2 for further details and performance comparisons between the two.

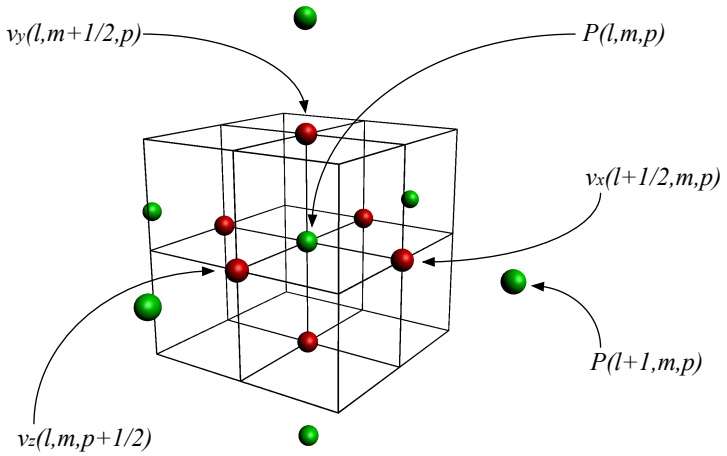


Figure 2.6 *Pressure and velocity node arrangement for the staggered grid in 3D.*

Numerical dispersion

The phase speed is the speed at which waves of a given wave number propagate in the finite difference scheme, and is defined as $\omega/|\beta|$. In the case of the wave equation, this should be constant for all wave numbers. However, for all but a few cases, FDTD schemes do not have a constant phase speed [42]. The result is termed *dispersion*, and minimising this effect is a key feature in any design [50] [51]. For some schemes, dispersion results in the higher frequency waves travelling slower than the correct velocity [52], although other designs can exhibit different behaviour.

Changes in phase speed can also vary with the direction of propagation, being *anisotropic* [53]. The standard finite difference scheme (2.14) shows maximum phase speed error along the coordinate line, and a minimum along the diagonals. Dispersion error also increases with the grid spacing, X, of the scheme, so choosing the stability bound gives the greatest accuracy (and the smallest allowable grid spacing).

Boundary conditions

Alongside the behaviour in the interior, a finite difference scheme must also describe the behaviour at the boundaries of the domain. This is of vital importance for simulations of acoustics, where the characteristics of the boundary have a dominant effect in the perception of the environment [54]. In electromagnetics the primary concern is reflectionless conditions, using techniques such as the Perfectly Matched Layer [55].

Reflectionless conditions are also useful in acoustic simulation, for instance in outdoor models and for instrument modelling in anechoic environments, but reflecting

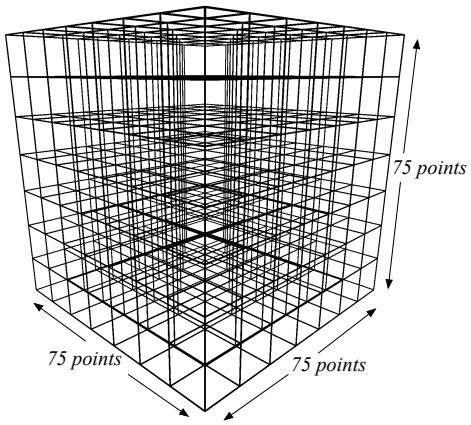


Figure 2.7 *A one cubic metre 3D grid at 44.1kHz requires 75 × 75 × 75 = 421,875 grid points.*

conditions are necessary to simulate natural room acoustics. The formulation of realistic, and stable, boundaries conditions in conjunction with minimising dispersion is the primary concern in the design of FDTD schemes for such models. This is an area of ongoing research, in terms of frequency dependent reflection and boundaries that conform to irregular geometries [52] [56] [48].

Computational cost

Whilst dispersion error and boundary conditions are major concerns when designing three-dimensional FDTD schemes, a further concern is the computational cost. Take the case of a simple one cubic metre space in three dimensions. At a sample rate of 44.1 kHz, the grid spacing at the Courant limit is 0.0135 m, the smallest spacing for which the scheme is stable (based on $c = 344m/s$). The grid required to mesh the cube is of size 75 × 75 × 75 = 421,875 grid points (Figure 2.7). The update equation that is applied to each grid point requires around ten floating-point operations, which means updating the grid requires 4.2 million floating-points operations.

In order to produce a single second of simulation output, the scheme needs to be computed for 44,100 time steps. This gives a total of 185 giga (billion) floating-point operations, just for a single second of output, for a single cubic metre. This level of computational cost is the primary motivation for this thesis. FDTD will be discussed further in terms of one-dimensional and two-dimensional schemes in Section 2.2.3.

2.1.5 Hybrid and alternative methods

As geometric methods apply well to high frequency behaviour, and wave-based models can suffer from numerical dispersion at high-frequency, there have been various attempts to combine the two in a hybrid approach. For example, combining FEM and ray-based methods [57], and FDTD with beam tracing and acoustic radiance [58].

Other approaches have been used to try to reduce the raw computational cost of full three-dimensional wave-based methods. For instance, the hybrid use of three-dimensional and two-dimensional systems [39], where early domain reflections are handled using a three-dimensional digital waveguide mesh, and two-dimensional system used for later diffuse effects. Domain decomposition approaches [59] [60] have also been considered.

2.2 Physical modelling synthesis

Whilst research concerning virtual acoustics focuses on three-dimensional propagation in rooms, *physical modelling synthesis* focuses on the simulation of musical instruments [61]. These two fields are not exclusive, as they merge at the point where full three-dimensional models of musical instruments are embedded in a virtual acoustic environment [62].

Digital sound synthesis can be traced back to the late 1950s with the development of wavetable oscillators and filters running on early computer systems. Abstract synthesis techniques such as additive, subtractive, and frequency modulation (FM) were conceived in the 1960s and 1970s [63], and led to the first digital synthesizer, the Yamaha DX7, in 1983 (see [64] and [35] for reviews of these techniques).

Although these methods may in some cases have aimed to create sounds that emulate real acoustic instruments, they are all based on abstract principles. Physical modelling synthesis seeks to create realistic acoustic sounds by starting with a purely physical description of the vibrations of the system itself. This typically results in a mathematical description using partial differential equations, which are then solved (or approximated) using some numerical method. Given an accurate mathematical description and a set of playable parameters, the resulting output waveforms should closely approximate those of real acoustic instruments.

This approach to sound synthesis is not new, being used in speech modelling by Kelly and Lochbaum in the early 1960s [65], and for strings by Ruiz in the late 1960s

[66] [67]. One of the first large-scale systems was developed during the 1970s, the CORDIS environment, which used mass-spring networks to model wave propagation [68]. Some more recent approaches to physical modelling synthesis are based on modal synthesis [69], the digital waveguide method [70] [71] [72], and the finite difference time domain method [73] [74].

2.2.1 Modal synthesis

Modal synthesis approaches physical modelling from the perspective of modal analysis, which is based on a decomposition of the dynamics of a vibrating object into modes, each of which oscillates at its own modal frequency. These modes are the patterns of motion that result from an excitation at a given frequency. Having performed the analysis, such a description can then be "rendered" to produce a simulation and audio output. The use of modal analysis for audio synthesis was originally developed at IRCAM [69], and became the basis of the MOSAIC (later Modalys) software.

Modal synthesis can be useful for modelling arbitrarily shaped objects [75], and is capable of very high quality resolution. However, such results may depend on a computationally expensive analysis process [76]. Despite this, research into modal synthesis is ongoing [77] [78].

2.2.2 Digital waveguides

Sound synthesis using digital waveguides is an important part of the history of physical modelling, as it led to a proliferation of research in the area, and also to the very first commercially available synthesizer that used this technique (the Yamaha VL1 released in 1994). This was made possible by the development of a highly efficient algorithm for computing one-dimensional wave propagation in objects such as strings and tubes, which was capable of producing real-time synthesis even on hardware from the early 1990s. The development of digital waveguides began with the work of Karplus and Strong in 1983 [79], with their efficient design for producing string tones based on a recursive filter. This algorithm produced harmonically rich tonal output for minimal computational cost. This work was then extended by Jaffe and Smith [80], which led to the generalised digital waveguide method for strings, which was then later applied to tubes for wind instruments [70] [81].

The core principle of the digital waveguide is the travelling wave solution to the one-dimensional wave equation, attributed to D'Alembert in the 18th century. The

lossless, linear, one-dimensional wave equation at position x at time t can be expressed as

$$y(x,t) = y_r(x - ct) + y_l(x + ct) \tag{2.24}$$

where c is the velocity of propagation, and $y(x,t)$ is the transverse displacement. Here y_r and y_l are the rightward and leftward travelling wave components. This can be viewed in the case of a plucked string, Figure 2.8. The initial string displacement decomposes over time into two travelling wave components. These components are reflected back with inverse phase at rigid terminating boundaries. The total string displacement at any given time is the sum of the two travelling wave components.

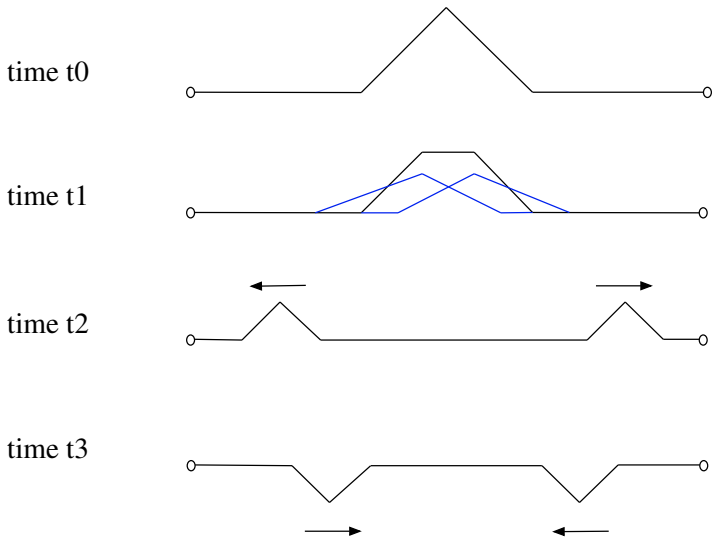


Figure 2.8 *Travelling waves on a plucked string. The string is plucked at time t0. By time t1 the solution begins to separate into the leftward and rightward travelling wave components. At time t2 these components are fully separated, and will reflect back with inverted phase at a rigid boundary termination.*

The key assumption in terms of implementing the digital waveguide is that the propagation of the travelling wave components in the string interior can be modelled as being lossless. The waves propagate through the interior without a change of shape or speed. Using this assumption, the plucked string can be modelled using two simple *delay lines*, one representing the leftward component and one the rightward. These delay lines are connected at either end, where boundary losses can be introduced. The output of the system is the sum of the two component delay lines at some point (Figure 2.9).

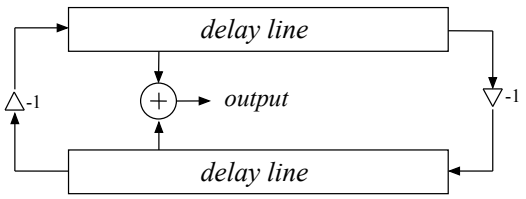


Figure 2.9 *Simple digital waveguide system for a plucked string.*

The system can be easily implemented on very basic hardware, as the delay lines are created using a standard computational structure, the *circular buffer*. In its most basic form, this consists of a read/write head that iterates over a defined segment of linear computer memory. The computational process is very simple; first the output of the delay line is read from the current memory location, then the new value is written in its place, and then the read/write head is incremented to point to the next memory location (Figure 2.10). This process continues with the head looping back to the start location from the end of the delay line.

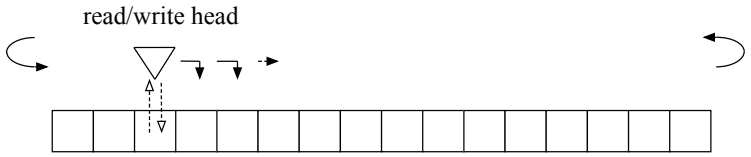


Figure 2.10 *Implementation of a delay line with a single head circular buffer. The read/write head first reads the value from the memory element (the output of the delay line), then overwrites it with the new value (the input to the delay line), and then shifts along to the next element in memory.*

This is extremely computationally efficient, with the only extra computation being the processing at the boundaries. It was this efficiency that allowed the technique to be used for real-time synthesis on DSP chips and desktop computer systems that were available in the late 1980s and early 1990s [70] [82]. The digital waveguide approach has also been applied to two-dimensional and three-dimensional systems, in the form of the digital waveguide mesh. However, these do not possess the same efficiency, when compared to finite different schemes, as holds for the one-dimensional case [73] [35].

2.2.3 Finite difference time domain method

Direct numerical simulation can be applied to systems that simulate the vibrations produced by musical instrument components, i.e strings, plates, acoustic tubes, and soundboards [83]. Whilst digital waveguides and modal synthesis both rely on simplifying assumptions for their efficiency, direct numerical simulation comes from a more analytical background. Methods such as finite difference approximations do not seek to simplify the systems, but provide a direct procedure for translating the mathematical descriptions into discrete systems that can be computed in an iterative manner.

The finite difference time domain method (FDTD) for one-dimensional and two-dimensional systems is analogous to that described for three-dimensional wave propagation in section 2.1.4. These types of schemes can be applied to a wide range of systems, including those exhibiting nonlinear behaviour, which is an important aspect in terms of high fidelity modelling of acoustic systems. The spatial domain is discretised into a grid of node points, representing some physical value such as displacement. The system is then updated over discrete time steps, computing the new values of the nodes based on those from previous time steps.

The use of finite difference methods for sound synthesis dates back to the late 1960s for simple one-dimensional systems [66]. Simulations of stringed instruments were developed by Chaigne and others for modelling guitars and pianos [74], including simulating the soundboard [84] [85]. Percussion instruments have also been attempted, based on bars, membranes and plates, both in terms of linear and nonlinear models [86]. See the text by Bilbao for a comprehensive treatment of FDTD applied to numerical sound synthesis [35].

Although the finite difference time domain method offers many advantages over other approaches, such as generality and simplicity, it must still address an issue that affects all direct numerical methods, that of computational cost. Even one-dimensional systems, such as multiple bars or strings, can require the majority of the processing power of a modern CPU to run in real-time at audio sample rates. Simple two-dimensional systems quickly fall outside the realm of real-time processing without the use of some form of acceleration technique. A further advantage of FDTD schemes, especially in explicit forms, is the very high level of *data independence* at each time step, which can be exploited by parallel programming concepts.

2.3 Parallel computing using graphics processing units

Microprocessors that use a single central processing unit (CPU) were introduced in the 1970s, and for thirty years the density of transistors on the die continued to increase with each new generation of processor. The clock frequencies and compute capability grew by around 50% each time, and so the programs they executed ran considerably faster without any refactoring of the code.

By the mid 2000s the transistor size had become so small that further increases in density led to significant increases in power consumption and heat dissipation [87]. Manufacturers began to increase the number of cores included on the CPU as a means of delivering improved compute capability. Dual-core, and then quad-core processors have become standard in processor design. This, alongside vector processing, as introduced parallel operations into mainstream computing. At around the same time, graphics processing units (GPUs) were rapidly advancing as massively parallel devices, to meet the demands of the computer gaming and film industries.

Although parallel computing has existed in the scientific community for decades, its use was largely restricted by the cost and availability of supercomputing hardware. Whilst laboratory supercomputers achieved teraflop performance only as recently as 1996 (the Intel ASCI Red) [88], the first teraflop GPU was released in 2008 (the AMD Radeon HD4800), just twelve years later [89]. GPUs are now capable of multi-teraflop performance, around ten times that of current generation CPU hardware (Figure 2.11). With these advances in GPU technology, the focus is now on parallel programming techniques to utilise such performance.

2.3.1 Parallel computing

Parallel computing is based on the principle of dividing a large computational problem into smaller subtasks, and then computing these subtasks in a concurrent manner [91]. Unlike sequential algorithms programmed for uni-processor architectures, there are a variety of models and hardware architectures available for parallel algorithms, such as shared-memory models, cluster systems, grid systems, and others. Parallel computing can be performed on hardware ranging from small devices such as mobile phones and tablets, through to the largest supercomputing systems. The primary motivation is to achieve improvements in either the run time or the quality of the results when compared to a sequential algorithm performing the same problem.

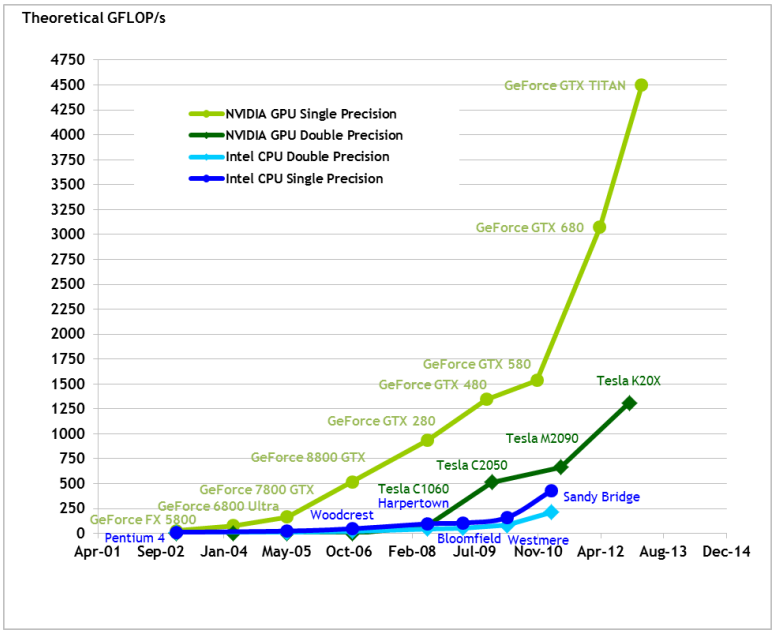


Figure 2.11 *Comparison of CPU and GPU flop rates, from Nvidia Corp. [90]*

Parallel algorithm designs

Due to differences in hardware architecture across parallel computing systems, the design of parallel algorithms depends on the hardware being employed. However, the high level methodology can be viewed as a four step process [92]:

1. *Partitioning*: Decompose the problem into as many subtasks as possible.
2. *Communication*: Analyse the communication requirements between these subtasks.
3. *Granularity control*: Reduce the level of communication by combining fine-grained tasks (those with a high frequency of communication) into larger coarse-grained tasks.
4. *Mapping*: Assign these coarse-grain subtasks to the available processors, and find the optimal balance between communication and the level of parallelism.

Ultimately, the efficiency of a particular algorithm implemented on given parallel hardware will depend on two aspects: the level of parallel computation, and the ability to transfer the data required to perform these calculations. This balance of *compute to memory access* is fundamental to all parallel computing, and is always a variable factor depending on both algorithm design and hardware.

Amdahl's and Gustafson's laws

The theoretical maximum speedup S that can be achieved by using N number of processors can be given by Amdahl's law

$$S(N) = \frac{1}{s + \frac{1-s}{N}} \tag{2.25}$$

where s is the fraction of the program that is serial [93]. Here, the elements of a program that cannot be made to execute in parallel have a large limiting effect on the available speedup. For example, if 99% of the program can be executed in parallel and 1% has to run in serial, a 100X speedup of the parallel section will result in a 50X speedup of the entire program. However, if only 30% of the program can be made parallel, then a 100X speedup of the parallel section only results in a 1.4X speedup overall, which is a huge reduction.

A variation on this principle was proposed by Gustafson, based not on fixed problem sizes but on solving the largest problem size possible in a practical amount of time [94]. This is expressed as

$$S(N) = N + (1 - N) \cdot s \tag{2.26}$$

This formula more accurately expresses the perception that increases in computational power lead to expected increases in the capabilities of the system.

Parallel computer models

The design of parallel computing hardware is based on a number of key issues [92].

- The type and arrangement of processors that are used, which may be based on some form of CPU, or more recently a combination of CPUs and GPUs.
- The efficiency of communication between these processors and with any global memory store.
- The ability to execute the same program over subsets of the data, or execute individual programs over the entire data.
- The level of asynchronous behaviour that is allowed.

This has led to a range of different architectural models for the hardware design. A standard classification of parallel computer architectures is that proposed by Flynn [95], which consists of four different categories:

- *Single Instruction, Single Data (SISD)*: A simple sequential computer, without a parallel architecture.
- *Single Instruction, Multiple Data (SIMD)*: A system that executes the same instruction on many different data elements, such as a vector processor.
- *Multiple Instruction, Single Data (MISD)*: A system where multiple processors use the same data and verify correctness, for example in safety critical systems.
- *Multiple Instruction, Multiple Data (MIMD)*: A system where multiple processors can execute different instructions on different data, using either a single shared memory space, or a distributed memory approach.

Whilst supercomputing systems typically use a MIMD architecture using distributed memory, individual graphics processing units consist of a massively parallel SIMD arrangement applied using threads (known as Single Instruction Multiple Thread, or SIMT). They may contain many hundreds of cores, each of which is itself a multithreaded processor (known as a *streaming processor*) that shares control logic and instruction cache with other cores.

2.3.2 Evolution of GPU computing

The use of graphics processing units in applications other than graphics rendering is not new. During the 1990s and 2000s, a small number of developers were using the GPU as a computing resource [96], but progress was made difficult due to the lack of any standard application programming interface (API) for the purpose. Programming the GPU to perform general purpose calculations required the use of graphics APIs such as DirectX and OpenGL [97]. This field at the time was known as *"GPGPU"*, or *general purpose programming for graphics processing units*.

Whilst there had been research into APIs specifically for GPU programming such as BrookGPU [98], it was not until the release of Nvidia's CUDA platform in 2006 that major progress was made. Alongside OpenCL that was released on 2008, these provided standardised methods for developers to interact with GPUs for general purpose computation. Although the Khronos Group and Apple Inc. continue to push OpenCL as a highly scalable and multi-platform language for parallel computing, the CUDA language has been adopted as the primary focus of the scientific community [99]. In terms of program development, both OpenCL and CUDA are based on keyword extensions to the ANSI C language with appropriate APIs.

2.3.3 GPU architectures

GPUs such as Nvidia's Tesla products are massively parallel processors consisting of many subsets of cores. Figure 2.12 shows the high level block diagram of a Kepler GK110 chip. The GK110 chip in the Tesla K20 card has 13 streaming multiprocessors (SMXs). Each SMX unit itself consists of 192 single-precision cores (streaming processors), which gives a total of 2,496 cores across the entire chip. These provide a maximum of 3.5 teraflops at single precision. Each SMX also contains 64 double-precision units, which provide over a teraflop of theoretical performance for double precision floating-point operations.

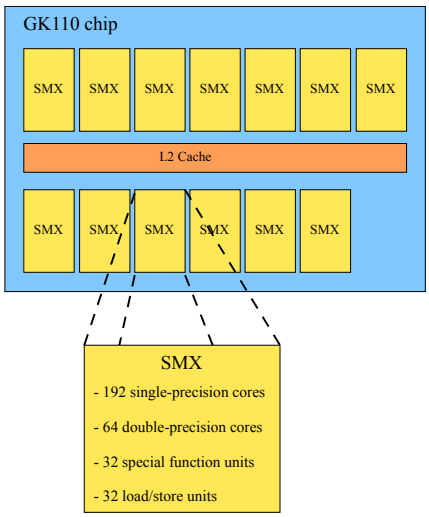


Figure 2.12 *The Nvidia GK110 chip, as used in the Tesla K20 device.*

Kernel launches schedule threads in groups of 32, called a *warp*. An SMX unit can accommodate a maximum of 64 warps simultaneously, which gives 2,048 threads. The chip is therefore capable of processing many thousands of threads concurrently, compared to tens of threads for a multi-core CPU. This level of parallelism is limited by the memory bandwidth, which on the K20 card is 208 GB/sec. For computations that have a low ratio of compute to memory access, the available memory bandwidth will be a limiting factor in overall performance.

Nvidia's GPU architectures have currently seen four major revisions, namely Tesla, Fermi, Kepler, and most recently Maxwell. The compute capability of a given hardware device describes the features of the architecture that are available, and are correspondingly named 1.x, 2.x, 3.x, and 5.x.

2.3.4 CUDA device memory

Nvidia's CUDA language allows developers to write code for both the *host* (the CPU), and on one or more *devices* (the GPUs). Program control executes in a sequential manner on the host, until a *kernel* launch is encountered. This initiates a number of *threads* that execute the given kernel on the device, whilst program control is handed back to the host. Whilst the host CPU code is written for a single memory store, GPU devices have four different types of memory which can be explicitly used in CUDA.

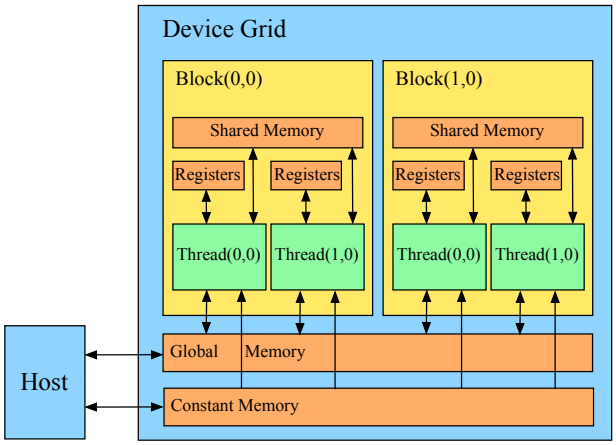


Figure 2.13 *Overview of CUDA device memory layout.*

At the outermost level, the host can both read and write data to the *global* memory on the device. This is the largest, and slowest, form of device memory, and is typically in the gigabyte range. This transfer is performed over the PCI bus, where data speeds are far less than access to the global memory by registers. The host can also write data to a small amount of *constant* memory, which is cached. Data is then consumed by blocks of threads, which have access to a *shared* memory resource, as well as local registers. Shared memory, which can only be accessed by threads in a given block, is in the range of kilobytes. This memory can also be configured around the L1 cache, and there is also a larger L2 cache serving the multiprocessors (Figure 2.12).

Individual threads can both read and write to global and shared memory, and can read from constant memory. The developer must explicitly allocate variables or pointers for the individual types of memory on the device, and the run time API provides functions for memory allocation, such as `cudaMalloc()`, and memory transfers. Memory management in CUDA is also somewhat related to the threading model. The indexing of threads is used to determine the access to data elements.

2.3.5 CUDA thread model

Although CUDA threads are scheduled to execute in *warps*, from a development perspective threads are grouped into *blocks*. The maximum number of threads in a block is 1,024 (for cards with compute capabilities of 2.0 through to 3.5). These thread blocks can be indexed according to one, two, or three dimensions. For example, a 3D thread block of size $4\times4\times2$ will have an index in dimension x ranging from 0 to 3, in dimension y ranging from 0 to 3, and in dimension z of 0 or 1 (see Figure 2.14).

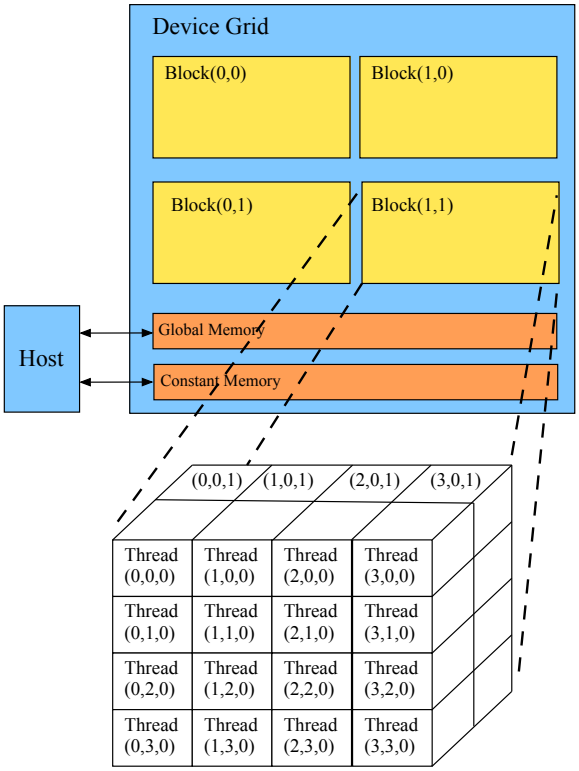


Figure 2.14 *CUDA thread grid and thread block indexing.*

The sizes of the dimensions are normally chosen as powers of two, so that some multiple of the x dimension is equal to the warp size. This is especially important in terms of optimising the memory bandwidth (see Section 2.3.6).

Given that a single thread block can only contain a maximum of 1,024 threads, issuing more than that number requires multiple thread blocks. These are organised into a *thread grid*, which again can be one, two, or three-dimensional, and thread blocks are indexed accordingly. As both the thread block and thread grid can be indexed in up to three dimensions, this gives multiple approaches for mapping threads to the data, especially for three-dimensional data sets (Section 3.5). The thread block represents an abstraction of the SMX on the chip, and threads an abstraction of the individual cores.

2.3.6 Performance optimisation

CUDA kernels are designed to be scalable, so that any kernel will function on any CUDA enabled device. However, obtaining optimal run times for the device requires a level of experimentation for the individual hardware being employed. Optimising kernel performance requires balancing the various constraints that apply to the system resources [97].

Memory bandwidth

Whilst a GPU typically has far greater memory bandwidth than a CPU, maximising the use of this bandwidth is still a critical issue. A kernel's performance can be measured in terms of its compute to memory access ratio (CMA). Many numerical algorithms, such as FDTD, have a very low CMA of around 1.0, meaning that there is a read or write to memory for every floating-point operation.

Consider such a kernel being executed on a Tesla K20 card, which has a memory bandwidth of 208 GB/sec. At single precision (four bytes) the maximum transfer rate is then 52 giga floats per second. With a CMA of 1.0, this gives a total flop rate of 52 gigaflops, far less than the theoretical maximum of 3.5 teraflops.

In order to achieve optimal memory bandwidth, it is vital to ensure that memory accesses are *coalesced*. When a warp of consecutively indexed threads attempts to load data in particular patterns from global memory (such as from consecutive elements), this transfer occurs in a single cycle rather than in individual calls [90]. This leads to substantial increases in data transfer rates, and is vital for delivering performance that is close to the theoretical maximum. In programming terms, this requires careful use of the thread IDs when accessing data in a kernel. The x dimension index (given in code as the `threadIdx.x`) should access data incrementally for consecutive threads.

Alongside memory coalescing in global memory transfers, exploiting shared memory is a further key optimisation. Reusing data stored in shared memory is far more efficient than repeatedly loading from global memory, as long as it can be used efficiently within a thread block. However, with the increases in caching levels for the Fermi, and later Kepler architectures, this has become less critical for certain types of algorithm [100]. Section 3.5 gives a performance comparison of shared memory versus caching for 3D FDTD schemes.

Thread execution

Each streaming multiprocessor has a number of resources that are dynamically assigned at run time, including registers and slots for threads and thread blocks. Each of these has a limited size, and so variations in the size of the thread block, or the number of variables assigned to registers, affects the overall run time of the kernels.

To achieve peak performance, high *occupancy* rates are required. The occupancy is the ratio of active warps to the maximum warps for a streaming multiprocessor. High occupancy rates help to hide the latency in accessing global memory. Varying the size of the thread block is a standard optimisation to find the best occupancy rates. A further technique is to vary the number of register variables that are used in the kernel. Either increasing or decreasing the number of variables in the kernel may have a beneficial effect, again depending on the size of the thread block.

The hardware executes instructions for all of the threads in the same warp before moving to the next instruction. The use of conditional statements in the kernel can lead to thread divergence, meaning that some threads follow a different instruction path. In this case, the hardware waits until all threads have completed, and the multiple paths lead to increases in the run time. Minimising thread divergence is of specific interest to finite difference schemes, where separate operations are performed on the interior and at the boundaries.

Additional methods

Further areas for manual code optimisation include methods such as data prefetching and improvements to the instruction mix. Data prefetching involved masking the loading of data from global memory to register by overlapping data access and computation. Instruction mix optimisation is where code is refactored to maximise the number of floating-point operations as opposed to addressing and branching. An example would be loop unrolling, which decreases loop iterations whilst increasing the number of floating-point calculations per iteration. These optimisations are again based on maximising the available memory bandwidth.

2.3.7 GPUs and 3D FDTD simulations

Three-dimensional numerical methods such as FDTD are constrained by two factors: the computation time, and the amount of memory required to hold the necessary data. Until recently, performing such simulations required the use of a large-scale cluster or

supercomputer system [101]. GPU technology has allowed a significant increase in the scale of simulations that can be achieved on desktop hardware. Teraflop performance and several gigabytes of memory have opened the door to large-scale numerical simulations in an economically viable manner. Supercomputers are of course still used [102], where greater levels of memory and compute capability are required.

GPUs were initially used to run two-dimensional FDTD simulations [103], quickly followed by three-dimensional models, in areas such as seismology and electromagnetics [104]. Much of the initial research focused on the use of shared memory, and varying the spatial stencil sizes of the schemes [105]. With the increased cache levels of the Fermi architecture cards, this led to comparisons of shared memory versus caching of global memory [106] [107].

GPUs have been used for acoustic modelling of mid to low frequencies in real-time schemes [108], and comparisons have been made between the digital waveguide mesh and FDTD implementations [109]. The simultaneous use of multiple cards has been demonstrated, although for three-dimensional schemes the scaling is not perfectly linear [110].

2.3.8 GPUs for audio processing

Wave-based acoustic simulations are not the only field where GPUs have been used to accelerate audio processing. Both modal synthesis [111], and more traditional additive synthesis techniques have been studied [112], achieving over a million sinusoids in a real-time application at audio sample rates. Two-dimensional FDTD systems have also been accelerated for use in a real-time system [113].

Fourier transforms are a key component of signal processing. The fast fourier transform algorithm uses a divide-and-conquer approach, which can be directly mapped to a parallel algorithm. Multi-dimensional FFTs can be accelerated using standard Nvidia libraries, CUFFT and CUFFTW.

GPUs are also beginning to be used in commercial digital audio applications, for example the Nebula3 VST plugin by Acustica Audio, where the core audio engine is accelerated using CUDA. See [114] and [115] for reviews of GPU implementations for various audio techniques.

2.4 Sparse linear algebra

The update equation for finite difference schemes can be written in a matrix form, where the state grids are arranged into vectors and the coefficients of the update equation are arranged into a matrix. For example, take the two-dimensional version of the scheme at (2.14), given as:

$$u_{l,m}^{n+1} = (2 - 4\lambda^2)u_{l,m}^{n} + \lambda^2(u_{l+1,m}^{n} + u_{l-1,m}^{n} + u_{l,m+1}^{n} + u_{l,m-1}^{n}) - u_{l,m}^{n-1} \qquad (2.27)$$

The state grids are decomposed into vectors by placing each column end-to-end, as shown in Figure 2.15.

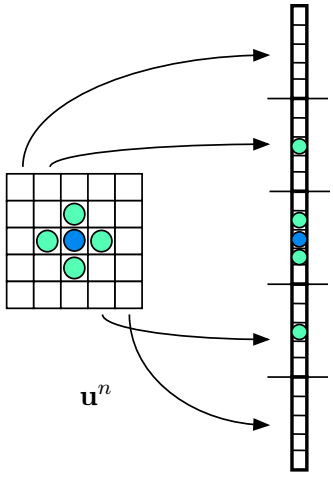


Figure 2.15 *Decomposition of a 2D grid into a 1D vector.*

Taking the updates for grid $\mathbf{u}^n$, these apply the coefficient $(2 - 4\lambda^2)$ to the centre point, and λ^2 to the neighbouring four points. These can be stored in a matrix, with the first coefficient along the diagonal, and λ^2 on separate bands (Figure 2.16).

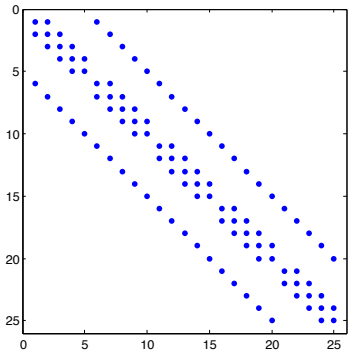


Figure 2.16 *Sparsity pattern of matrix* B *for a 2D system.*

A matrix by vector multiplication $\mathbf{B} \cdot \mathbf{u}^n$ will then compute the necessary update. The update equation (2.27) can then be written as

$$\mathbf{u}^{n+1} = \mathbf{B}\mathbf{u}^n + \mathbf{C}\mathbf{u}^{n-1} \tag{2.28}$$

where $\mathbf{B}$, and $\mathbf{C}$ are update matrices of coefficients, and are diagonally banded and therefore sparse. A more generalised version is given by

$$\mathbf{A}\mathbf{u}^{n+1} = \mathbf{B}\mathbf{u}^n + \mathbf{C}\mathbf{u}^{n-1} \tag{2.29}$$

For explicit schemes the matrix $\mathbf{A}$ is the identity matrix, but for implicit schemes this is not the case and solving for $\mathbf{u}^{n+1}$ requires a linear system solution at each time step of the simulation.

Complex physical modelling systems based on FDTD can use a combination of one, two and three-dimensional schemes, and matrix formulations are useful for unifying the design. In the case of some implicit schemes, a matrix form is necessary to store the coefficients that are constructed at each iteration in time. These schemes are useful as they often lead to greatly reduced numerical dispersion.

2.4.1 Definitions

The following standard notation is used to describe vectors and matrices, along with the basic operations that can be performed [116].

Notation

Let $\mathbb{R}$ be the set of real numbers. The vector space of all m-by-n real matrices is denoted by $\mathbb{R}^{m \times n}$ where

$$\mathbf{A} \in \mathbb{R}^{m \times n} \iff \mathbf{A} = \begin{bmatrix} A[1,1] & \dots & A[1,n] \\ \vdots & & \vdots \\ A[m,1] & \dots & A[m,n] \end{bmatrix} A[i,j] \in \mathbb{R}$$

A capital letter is used to indicate a matrix, for example $\mathbf{A}$, and $A[i,j]$ refers to the entry at (i,j). For vectors, let $\mathbb{R}^n$ denote the vector space of real n-vectors such that

$$\mathbf{x} \in \mathbb{R}^n \iff \mathbf{x} = \begin{bmatrix} x[1] \\ \vdots \\ x[n] \end{bmatrix} x[i] \in \mathbb{R}$$

Vector and matrix operations

Some basic operations over vectors and matrices are defined as:

the dot product:

$$c = \mathbf{x}^T\mathbf{y} \qquad \Longrightarrow \qquad c = \sum_{i=1}^{n} x[i]y[i]$$

saxpy (scalar ax plus y):

$$\mathbf{y} = a\mathbf{x} + \mathbf{y} \qquad \Longrightarrow \qquad y[i] = y[i] + ax[i]$$

matrix by vector multiplication ($\mathbb{R}^{m\times n} \times \mathbb{R}^m \to \mathbb{R}^m$):

$$\mathbf{y} = \mathbf{A}\mathbf{x} \qquad \Longrightarrow \qquad y[i] = \sum_{j=1}^{n} A[i,j]x[j]$$

gaxpy (general) ($\mathbb{R}^{m\times n} \times \mathbb{R}^m \to \mathbb{R}^m$):

$$\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{y} \qquad \Longrightarrow \qquad y[i] = y[i] + \sum_{j=1}^{n} A[i,j]x[j]$$

scalar-matrix multiplication ($\mathbb{R} \times \mathbb{R}^{m\times n} \to \mathbb{R}^{m\times n}$):

$$\mathbf{C} = \alpha\mathbf{A} \qquad \Longrightarrow \qquad C[i,j] = \alpha A[i,j]$$

matrix multiplication ($\mathbb{R}^{m\times p} \times \mathbb{R}^{p\times n} \to \mathbb{R}^{m\times n}$):

$$\mathbf{C} = \mathbf{A}\mathbf{B} \qquad \Longrightarrow \qquad C[i,j] = \sum_{k=1}^{p} A[i,k]B[k,j]$$

matrix addition ($\mathbb{R}^{m\times n} \times \mathbb{R}^{m\times n} \to \mathbb{R}^{m\times n}$):

$$\mathbf{C} = \mathbf{A} + \mathbf{B} \qquad \Longrightarrow \qquad C[i,j] = A[i,j] + B[i,j]$$

2.4.2 Sparse matrices

A *sparse* matrix is a matrix where the majority of the elements have a value of zero. The number of non-zeros (*nnz*) gives the density of the matrix. Whilst full matrices are stored in computer memory using either a row or column decomposition of the two-dimensional array, sparse matrices require a different arrangement in order to be efficient.

Sparse matrix storage systems

The simplest storage system is known as *triplet* form [117], or coordinate list (COO). This is simply a list of the non-zero elements with their associated (i, j) positions stored in two arrays of integers. Many other sparse storage systems exist, most of which are designed to further reduce memory requirements by exploiting the structure of the matrices. This is important for finite difference schemes, where the matrix representation is typically highly structured in the form of multiple bands. See Section 6.1 for details regarding the CSR, DIA and ELLPACK systems.

Sparse matrix computations

The purpose of using a sparse matrix storage system is both to reduce the memory requirement and to reduce the overall computational complexity. For example, matrix by vector multiplication reduces to $O(n)$ for a very sparse matrix, rather than $O(n^2)$. However, the type of sparse storage system used can have a significant impact on the implementation of matrix operations, both in terms of the design and performance efficiency.

Consider the matrix by vector multiplication $\mathbf{y} = \mathbf{A}\mathbf{x}$. There are two different approaches to computing this operation on a general matrix, row-based or column-based [116]. The row-based algorithm takes each row in turn, and iterates over the elements of that row along with the elements of $\mathbf{x}$. The products are summed into one element of the vector $\mathbf{y}$, as shown in Figure 2.17.

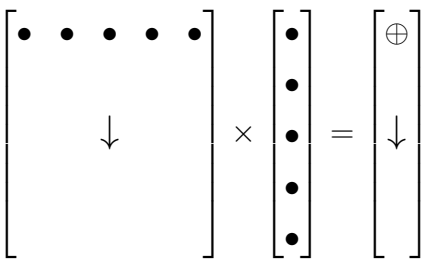


Figure 2.17 *Row-based matrix by vector multiplication.*

As an algorithm, this can be described as follows:

Algorithm 1 *Row-based matrix by vector multiplication.*

1: **for** $i = 0 : m - 1$ **do** ▷ Loop over the rows
2: **for** $j = 0 : n - 1$ **do** ▷ Loop over the columns
3: $sum \leftarrow sum + A[i, j]x[j]$
4: **end for**
5: $y[i] \leftarrow sum$
6: **end for**

The equivalent column-based version takes a different approach. The computation takes each column in turn. It iterates over the elements of that column, using a single value from $\mathbf{x}$, and summing each product into corresponding elements of $\mathbf{y}$.

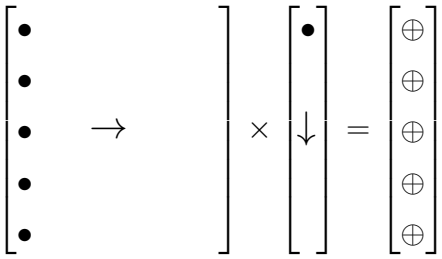


Figure 2.18 *Column-based matrix by vector multiplication.*

As an algorithm, the only difference from the row-based approach is that the order of the loops is switched and the elements of $\mathbf{y}$ are not overwritten. The outer loop is now over the columns, rather than the rows. In terms of dense matrix computations, this has very little difference. However, for sparse matrices the implementation of these two versions is entirely different, due to the structure of the storage system being employed. To implement the column-based approach using the CSC format is trivial,

Algorithm 2 *Column-based matrix by vector multiplication.*

```
1: for j = 0 : n − 1 do                          ▷ Loop over the columns
2:     for i = 0 : m − 1 do                      ▷ Loop over the rows
3:         y[i] ← y[i] + A[i, j]x[j]
4:     end for
5: end for
```

as the compressed pointer data gives the starting element of each column. However, implementing the row-based method in CSC would be inefficient as accessing consecutive elements of a given row is a complex operation. In general, the implementation of sparse matrix operations greatly depends on the storage system [118]. This also has implications for the parallel computation of matrix operations, as discussed in Chapter 6.

2.4.3 Solution to systems of linear equations

Solving for the unknowns in a set of linear equations is one of the most fundamental tasks in scientific computing [119]. This is expressed in matrix form as

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{2.30}$$

where $\mathbf{A}$ is a matrix of coefficients and $\mathbf{x}$ the vector of unknowns. The most efficient method used to solve for $\mathbf{x}$ depends on the properties of $\mathbf{A}$.

Matrix properties

It is useful to describe some common properties of matrices.

Square matrices:

$$m = n$$

Symmetric matrices:

$$\mathbf{A}^T = \mathbf{A}$$

Upper triangular matrices:

$$A[i, j] = 0, \qquad i > j$$

Diagonally dominant matrices:

$$|A[i,i]| \geq \sum_{j \neq i} |A[i,j]|, \quad \textit{for all } i$$

Positive-definite matrices:

$$\mathbf{u}^T \mathbf{A} \mathbf{u} > 0, \qquad \forall \mathbf{u} \in \mathbb{R}^n, \mathbf{u} \neq 0$$

(*if the matrix is real, then it must also be symmetric*)

Direct methods

For matrices that are square and unstructured the most common approach is a *direct* method. This consists of some form of elimination by pivoting the matrix along its diagonal such that it is reduced to a triangular form. An example would be *Gaussian* elimination. Such LU decomposition is itself of order $O(n^3)$. Solving the resulting triangular system is easily achieved using forward and back substitution (see Algorithms 3 and 4). For matrices that are symmetric and positive-definite, there are more efficient algorithms such as *Cholesky factorization* that produce a lower triangular matrix with positive entries on the diagonal [117].

Algorithm 3 *Forward substitution for Ly = b.*

```
1: for i = 0 : n − 1 do
2:     y[i] ← b[i]
3:     for j = 0 : i − 1 do
4:         y[i] ← y[i] − L[i, j] · y[j]
5:     end for
6:     y[i] ← y[i]/L[i, i]
7: end for
```

Note that the algorithms for forward and back substitution are not data independent over rows. Each element of the solution vector must be computed before the outer loop proceeds to the next, as the inner loop j counter is used to traverse the previously calculated values. This has major implications when working with sparse matrices and parallelisation using GPUs.

Algorithm 4 *Back substitution for Ux = y.*

1: **for** $i = n - 1 : 0$ **do**
2: $x[i] \leftarrow y[i]$
3: **for** $j = i + 1 : n - 1$ **do**
4: $x[i] \leftarrow x[i] - U[i, j] \cdot x[j]$
5: **end for**
6: $x[i] \leftarrow x[i]/U[i, i]$
7: **end for**

Iterative methods

Whilst direct methods arrive at the solution in a given finite number of operations, the *iterative* methods start with some approximation to the solution and modify this over steps until a stopping condition (convergence) is met. Various categories of iterative methods exist, such as stationary, Krylov subspace, Multigrid, and domain decomposition methods [120]. The Jacobi stationary method, and the Conjugate Gradient Krylov subspace methods are described here.

Jacobi iteration

The Jacobi method is one of a number of techniques based on decomposing the matrix into simple components such as the diagonal, or the upper and lower triangular sections. Other techniques include Gauss-Seidel and Successive Over Relaxation (SOR). The methods can also be applied in block form, or used as preconditioners.

The Jacobi iteration decomposes $\mathbf{A}$ into its diagonal ($\mathbf{D}$) and the remainder ($\mathbf{R}$) where $\mathbf{A} = \mathbf{D} + \mathbf{R}$. The algorithm then computes the following for k iterations:

$$\mathbf{x}^{k+1} = \mathbf{D}^{-1}(\mathbf{b} - \mathbf{R}\mathbf{x}^{k}) \tag{2.31}$$

This algorithm is useful from a parallel architecture perspective, as the majority of the computation is based on a matrix by vector multiplication. However, the Jacobi iteration will generally only converge when the matrix $\mathbf{A}$ is diagonally dominant [120].

Conjugate gradient method

For more generalised cases, the Krylov subspace methods are an important class of iterative method. These are based on projection processes, and include methods such as Minimum Residual, Generalized Minimum Residual Method (GMRES), Conjugate

Gradient, and Biconjugate Gradient (BCG). The Conjugate gradient algorithm is described in Algorithm 5.

Algorithm 5 *Conjugate Gradient method*

1: $\mathbf{r}_0 \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}_0$
2: $\mathbf{p}_0 \leftarrow \mathbf{r}_0$
3: **for** $j = 0, 1, ..., until\ convergence$ **do**
4: $\quad \alpha \leftarrow \mathbf{r}_j^T \mathbf{r}_j / (\mathbf{p}_j^T \mathbf{A}\mathbf{p}_j)$ ▷ α is a scalar
5: $\quad \mathbf{x}_{j+1} \leftarrow \mathbf{x}_j + \alpha_j \mathbf{p}_j$
6: $\quad \mathbf{r}_{j+1} \leftarrow \mathbf{r}_j - \alpha_j \mathbf{A}\mathbf{p}_j$
7: $\quad \beta \leftarrow \mathbf{r}_{j+1}^T \mathbf{r}_{j+1} / (\mathbf{r}_j^T \mathbf{r}_j)$ ▷ β is a scalar
8: $\quad \mathbf{p}_{j+1} \leftarrow \mathbf{r}_{j+1} + \beta_j \mathbf{p}_j$
9: **end for**

The major operations are two matrix by vector multiplications, alongside various vector operations such as the dot products and additions. The algorithm in this form suffers from a lack of robustness, as convergence is by no means guaranteed. Both the robustness and efficiency (the number of iterations required to converge) can be improved by using *preconditioning* [120].

Preconditioning consists of finding a suitable matrix $\mathbf{M}$, which is related to the matrix A and it should be easy to compute $\mathbf{M}\mathbf{x} = \mathbf{b}$. The resulting system, when applying the left-sided preconditioner, is:

$$\mathbf{M}^{-1}\mathbf{A}\mathbf{x} = \mathbf{M}^{-1}\mathbf{b} \tag{2.32}$$

The preconditioned matrix is then used in the iterative algorithm along with the matrix $\mathbf{A}$, for example with the Conjugate Gradient method, to give a preconditioned version (PCG). The choice of preconditioner can range from a simple Jacobi form (the diagonal), to factorizations such as incomplete Cholesky, to sparse approximate inverses [121] and others. Important considerations include issues such as the sparsity of the preconditioner (which will affect the computational requirements), as well as the ability to implement the inverse system on a parallel architecture. The incomplete Cholesky factorization is a commonly used approach, which shows useful convergence properties for many types of systems.

2.4.4 Sparse linear algebra and GPUs

Accelerating linear algebra operations is one of the key areas of research in high performance computing (HPC). The expansion in the use of GPUs for general purpose computation has led to research into both dense and sparse matrix operations using this hardware. This section reviews libraries and related research for sparse operations.

GPU accelerated sparse matrix libraries

There are several libraries currently available for performing sparse matrix operations on GPUs, such as Nvidia's CUSPARSE, the CUSP library [122], and CULA [123]. The CUSPARSE library is part of the CUDA toolkit, and provides basic linear algebra functionality as well as more specialised operations such as triangular systems solvers. The API is consistent with other Nvidia libraries. However, the functions only make use of either CSR format, or an opaque hybrid form using ELLPACK and COO.

The CUSP library is more flexible in terms of storage systems, including DIA and ELLPACK systems. However, the API for the CUSP library is abstracted a long way from the underlying CUDA operations, leading to difficulties in implementing algorithms that require extensive control over the GPU memory. CULA offers many routines for performing iterative linear system solutions, but offers only the CSR format.

Sparse matrix by vector multiplication

The matrix by vector product is a central component of iterative linear solvers, but achieving high levels of floating-point throughput on the GPU can be difficult due to the memory access patterns. Therefore, the design of the sparse matrix storage system, as well as exploiting matrix structure, is critical in optimising performance.

Structured grid computations have been studied in [124], using block-diagonal systems, and a framework for experimenting with new storage systems is detailed in [125]. As caching levels have increased with first the Fermi and later the Kepler architecture GPUs, the impact on vector-matrix operations has been studied in [126] and [127].

Sparse linear system solutions

Forward and back substitution for dense matrices has many possibilities for parallel execution, performing concurrent calculations at each row [128]. In the case of sparse matrices the number of elements in each row can become very small, and so the available parallelism reduces [129] [130].

Further concurrency can be found using a *level scheduling* approach, which is used in the triangular solvers in Nvidia's CUSPARSE library [131]. This approach is based upon grouping unknowns in $\mathbf{x}$ into different levels so that all unknowns in the same level can be computed simultaneously. In the best case, where the number of levels is one, this corresponds to the diagonal matrix where each row is independent. In the worst case, where the number of levels reaches the number of rows, the algorithm is completely serial.

However, the efficiency is still very dependent on the structure of the matrices. Even at its most efficient, the achievable speedups are in the range of 3X to 4X over a CPU implementation [132]. In the many cases a CPU version can achieve better efficiency than a GPU version [133]. For iterative solvers such as Preconditioned Conjugate Gradient (PCG) it can be useful to find preconditioner matrices such as sparse approximations [134] [135], where $\mathbf{x} = \mathbf{M}^{-1}\mathbf{b}$ can be computed in a parallel manner using a matrix by vector multiplication.

2.5 Timing methods

Much of this thesis is concerned with performance efficiency, comparing the length of time that various algorithm implementations take to execute on either CPU or GPU hardware. For CPU codes the functions from the `time.h` header file in the standard C library are used. The `clock( )` function gives the processor tick count, and can be converted to seconds by dividing by the `CLOCKS_PER_SEC` macro. This gives millisecond timing information for single threaded code, but is not reliable for multi-threaded code. The `time( )` function is used instead, giving timings accurate to the nearest second as it references the system wall clock time. As the majority of the CPU codes detailed in this thesis execute over many minutes (or indeed hours), this provides sufficient accuracy for the purpose of comparisons and computing speedups.

For timing GPU codes, the `clock( )` function or `time( )` function can also be used [136], called from the host code. However, as CUDA kernel launches execute asynchronously with respect to the host, a `cudaDeviceSynchronize( )` instruction must be called to ensure that the device has completed all thread execution before obtaining a timing result. This function blocks the CPU thread until all CUDA calls previously issued are completed. The codes are generally executed over a large number of time iterations, for example 44,100, and so a single `time( )` is used before and then after the time iteration loop. For more complex codes where there are multiple kernel

calls at each iteration, the Nvidia command-line profiler is used to detail individual kernel timings, using `export CUDA_PROFILE=1`. As there is some variation in timing results for a given simulation (especially on the GPU), all stated times are the average of three runs of the simulation.

Part II

Accelerating virtual acoustics

Chapter 3

Computing solutions to the 3D wave equation

This chapter examines the computation of the basic scheme for the three-dimensional wave equation. A test case simulation is used to demonstrate parallel programming techniques and to compare CPU performance to that of the GPU. Multi-threaded code written in the C language is used to evaluate the maximum efficiency of the CPU and provide realistic benchmark figures at both single and double precision floating-point arithmetic. This is then compared to the performance of GPU codes that use the CUDA language, along with various optimisation strategies to achieve the best performance on single and also multiple GPU devices.

3.1 The basic 3D scheme

The simplest form of finite difference scheme for the 3D wave equation over a cubic grid is the reduced form where $\lambda = 1/\sqrt{3}$. This is given by

$$\Psi^{n+1}_{l,m,p} = \frac{1}{3}(\Psi^{n}_{l+1,m,p} + \Psi^{n}_{l-1,m,p} + \Psi^{n}_{l,m+1,p} + \Psi^{n}_{l,m-1,p} + \Psi^{n}_{l,m,p+1} + \Psi^{n}_{l,m,p-1}) - \Psi^{n-1}_{l,m,p} \tag{3.1}$$

as detailed in Section 2.1.4.

Each grid point in $\mathbf{\Psi}^{n+1}$ is updated using the six nearest neighbouring points from $\mathbf{\Psi}^{n}$ and the centre point from $\mathbf{\Psi}^{n-1}$, as shown in Figure 3.1. For this simple test case zero boundary conditions are employed (the value at the boundary is fixed to zero).

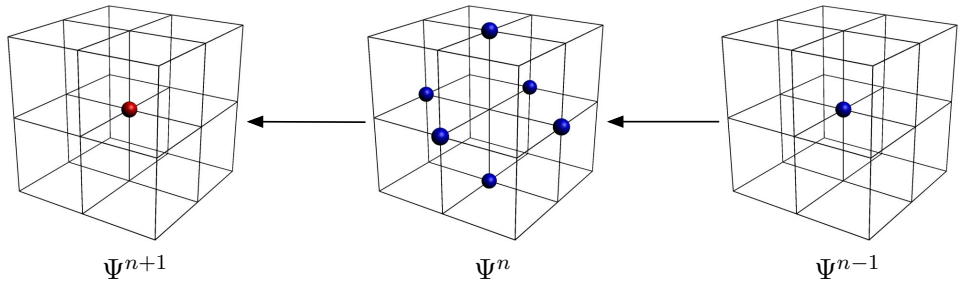
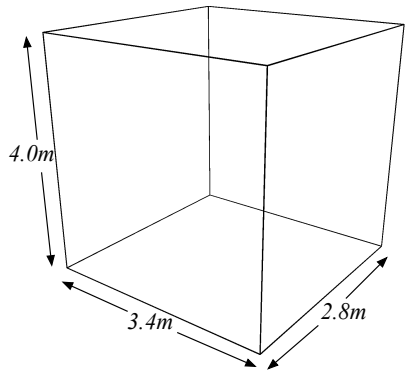


Figure 3.1 *Update nodes used for the basic 3D scheme.*

3.2 Test simulation

The test simulation computes this scheme over a domain using 15.7 million grid points, as shown in Figure 3.2. At a sample rate of 44.1kHz the grid spacing is 13.51mm, giving a physical domain of size 38m^3. Throughout this chapter N_l, N_m, and N_p denote the number of grid points in each of the dimensions X, Y, and Z.



N_l = 256 points = 3.4m
N_m = 296 points = 4.0m
N_p = 208 points = 2.8m

Figure 3.2 *Test case simulation of size 256 × 296 × 208 = 15,761,408 grid points. At 44.1kHz this represents a domain of 3.4m × 4.0m × 2.8m = 38m^3.*

The scheme is computed for 44,100 time steps in each test, representing a one second time simulation. A short raised cosine impulse is used as an input, which is summed into the grid at a given point as a 'soft' source [137]. To verify correctness, the value at a grid point is read at each iteration in time, and stored in an output array. In terms of computational cost, there are 15.7 million grid points which are each updated 44,100 times, giving a total of 695 billion updates using equation (3.1). This test simulation is used throughout this chapter for both CPU and GPU codes.

3.3 Linear decomposition of three-dimensional data

The three-dimensional data grids must be held in memory and addressed using some form of array notation. Although the C language allows three-dimensional arrays to be created and referenced using notation such as `array[p][m][l]`, this is not used here as CUDA requires the explicit allocation of linear memory. Instead, one-dimensional arrays are created and the three-dimensional data is mapped onto the space. This can be achieved using either a row-major or column-major approach. For row-major, each $N_l \times N_m$ layer of the N_p dimension is decomposed by placing every row side-by-side in linear memory, as shown in Figure 3.3.

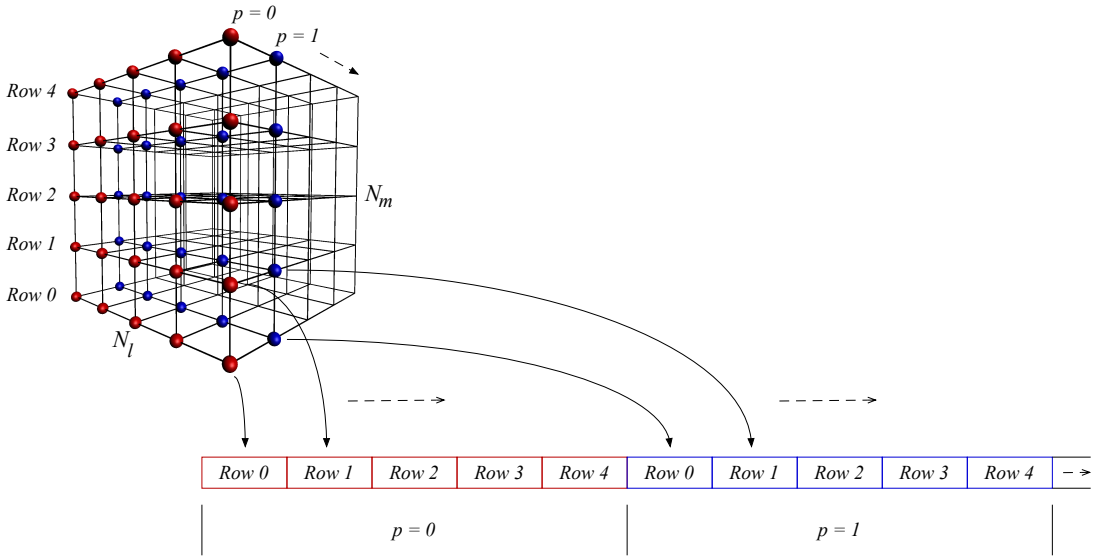


Figure 3.3 *Linear decomposition of 3D memory into a linear array using a row-major format. Each N_p layer is decomposed by row, and then placed in series.*

The data is then addressed using the following system

$$data(l, m, p) = linearData(p \cdot N_l \cdot N_m + (m \cdot N_l) + l) \tag{3.2}$$

This requires five floating-point operations to compute the linear position. However, in practice this address need only be computed once for a given update, as the neighbouring points can be accessed by simple shifts away from the centre point.

3.4 CPU benchmarks

To provide a realistic benchmark for comparisons to GPU performance it is essential to demonstrate the maximum performance achievable using a CPU. As current CPUs have multiple cores, some form of parallel programming is necessary to make use of all available cores. This is achieved using multi-threading.

3.4.1 Single thread code

The structure of a single thread code for the test simulation is shown in Algorithm 6 (the full C code is given in Appendix B.1).

Algorithm 6 *Single thread algorithm for the basic 3D scheme, zero-based indexing.*

```
1:  Setup parameters
2:  Create memory
3:  for n = 1 : NF do                                  ▷ Loop over the time steps
4:      for p = 1 : N_p − 2 do                         ▷ Loop over the spatial dimensions,
5:          for m = 1 : N_m − 2 do                     ▷ excluding the boundary elements
6:              for l = 1 : N_l − 2 do
7:                  Update node Ψ^{n+1}_{l,m,p}
8:              end for
9:          end for
10:     end for
11:     Update the input and read the output
12:     Swap data pointers to move forward in time (i.e. Ψ^{n+1} becomes Ψ^n)
13: end for
14: Print output
15: Free the memory
```

Whilst the scheme uses data from both one and two previous time steps, the implementation requires only two data grids. As only the centre point from $\Psi^{n-1}_{l,m,p}$ is required for each update, an overwriting process can be used to read the value before replacing it with the new value of $\Psi^{n+1}_{l,m,p}$. This is important when considering large-scale simulations, where available memory is a limiting factor. The test case uses 126MB of memory for each of the two grids at double precision (252MB in total).

3.4.2 Multi-threaded code

POSIX threads (or pthreads) are a standard mechanism for issuing a sub-program that will execute with multiple instances [138]. Each thread performs a set of simultaneous instructions over shared data, and so the thread ID is used to allocate parts of the data set to individual threads.

The basic wave equation scheme is naturally data independent over each time step. The computation can therefore be parallelised at each iteration by simply partitioning the data domain. For example, if four threads are issued then each thread will compute the update equation for one quarter of the linear memory (Figure 3.4). This represents partitioning the three-dimensional data over the N_p dimension layers.

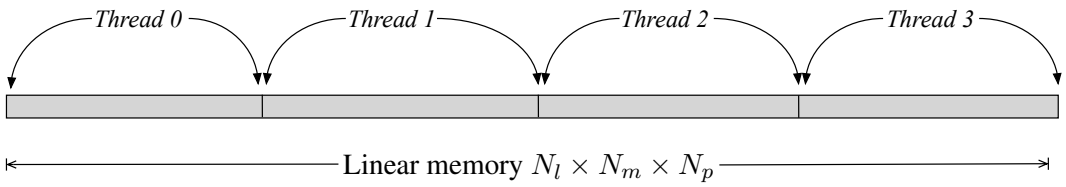


Figure 3.4 *Domain partitioning using four threads.*

The algorithm to compute this has a two-part structure: the main program, and a thread kernel. Algorithm 7 shows the new time iteration loop. The nested `FOR` loops over the spatial dimensions are replaced by a loop over the number of threads.

Algorithm 7 *POSIX thread main program for the basic 3D scheme.*

```
1: Setup parameters
2: Create memory
3: for n = 1 : NF do                          ▷ Loop over the time steps
4:     for i = 1 : numberofthreads do
5:         Launch thread kernel
6:     end for
7:     Wait for threads to finish             ▷ Synchronization barrier
8:     Update the input and read the output
9:     Swap data pointers
10: end for
11: Print output and free memory
```

After issuing the threads, the main program must wait until all threads have completed before swapping the data pointers. The thread kernel is described in Algorithm 8. Each thread is allocated a thread ID number, and this is used to determine the range of N_p dimension layers to update. The full code is given in Appendix B.2.

Algorithm 8 *POSIX thread kernel for the basic 3D scheme, zero-based indexing.*

```
1: Obtain thread ID number
2: Determine start N_p layer for this thread, ps
3: Determine end N_p layer for this thread, pe
4: for p = ps : pe do                          ▷ Loop over range of N_p layers
5:     if p > 0 and p < N_p − 1 then           ▷ Check not at the boundary layer
6:         for m = 1 : N_m − 2 do
7:             for l = 1 : N_l − 2 do
8:                 Update node Ψ^{n+1}_{l,m,p}
9:             end for
10:        end for
11:    end if
12: end for
```

3.4.3 Performance evaluation

The single and multi-threaded codes were tested on two different CPU hardware: an Intel i7 3770S processor which has four cores, and two Intel Xeon E5-2620 processors which have six cores each. Note that these processors employ 'hyper-threading' to allow two threads to execute on each core; see Appendix A.1 for the full hardware specifications. The GCC compiler is used in all cases and the code was tested at first with no compiler optimisation (using -O0), and then using -O3 optimisation. The later is the compiler's most aggressive level of optimization. The results for the single thread code for both the i7 and Xeon processors are shown in Table 3.1.

Version	**Processor**	**Single prec.** (min:sec)	**Double prec.** (min:sec)
Single thread -O0	Xeon	143:18	144:54
Single thread -O0	i7	82:33	83:12
Single thread -O3	Xeon	37:15	38:54
Single thread -O3	i7	21:01	23:52

Table 3.1 *CPU results for the basic 3D scheme at single and double precision.*

In the worst case, with no optimisation on the Xeon processor, the simulation takes nearly two and half hours to complete. The i7 is considerably faster due to its higher clock rate, completing in one hour and twenty minutes. Switching on -O3 compiler optimisation results in major improvements, with close to 4X speedups on both processors. This gives simulation times of between twenty to forty minutes.

The compiler optimisation is invoking vectorised operations, either SSE or AVX instructions, which perform floating-point calculations on sequential data elements simultaneously. Manually unrolling the inner loop over the N_l dimension into steps of two, four or eight resulted in no further efficiency gains. The -O3 optimisation is used for all further tests. At this stage, there is very little difference between single precision and double precision arithmetic. The multi-threaded code was tested for a range of thread sizes on each processor, as shown in Table 3.2.

Version	**Processor**	**Single prec.** (min:sec)	**Double prec.** (min:sec)
2-threads	i7	16:25	18:11
4-threads	i7	11:52	21:14
8-threads	i7	10:15	23:02
16-threads	i7	10:14	22:47
2-threads	Xeon	19:16	19:45
4-threads	Xeon	9:51	10:22
8-threads	Xeon	6:22	7:34
16-threads	Xeon	6:08	10:57

Table 3.2 *Multi-threaded CPU results for the basic 3D scheme at single and double precision.*

The results vary according to the processor, and the precision level. For the i7, using 8 or 16 threads achieved a 2X speedup over the optimised single thread code, but only at single precision. At double precision, only the use of 2 threads shows any sizeable gains at all. The Xeon processor behaves differently. Running 16 threads gives a 6X speedup at single precision, and a 4X speedup at double, with elapsed times of 6 minutes and 11 minutes respectively. From being slower than the i7 when running the single thread code, it is now performing nearly twice as efficiently.

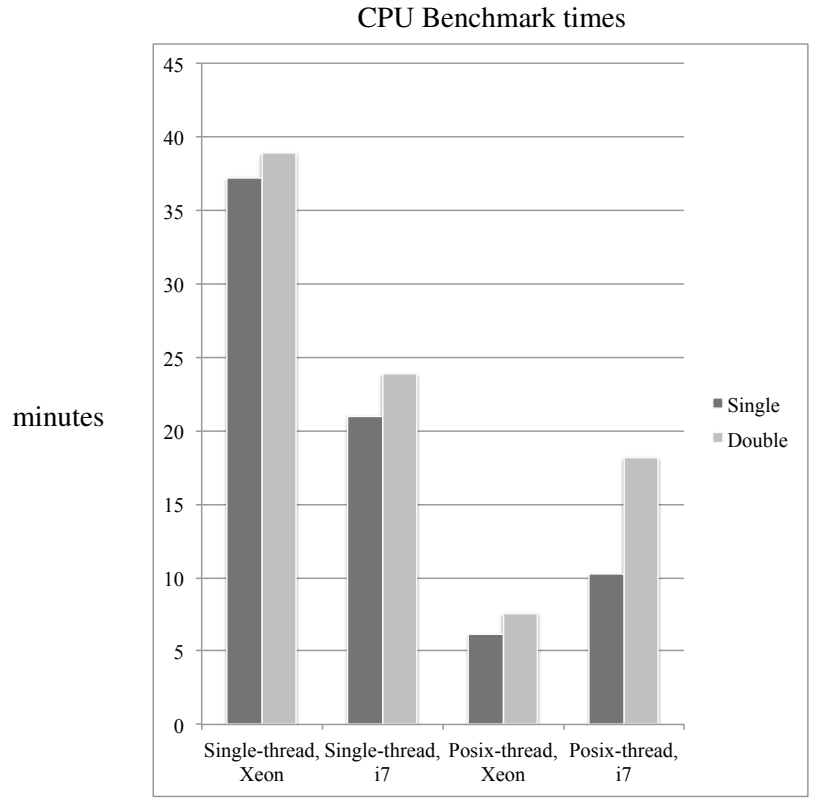


Figure 3.5 *CPU benchmarks for the basic 3D scheme at single and double precision*

Figure 3.5 plots the most efficient times for each of the tested CPU systems. From starting with an unoptimised single thread code running in two and half hours on the Xeon, the most efficient simulation now runs in just six minutes. This is a 24X speedup simply by optimising the code for the CPU. This clearly demonstrates the multi-core capacity of these processors, and the need to maximise CPU potential when performing benchmarking.

3.5 GPU kernel design and optimisation

In the previous section a small number of threads were used to update parts of the data grid in parallel. To make use of a GPU device, the CUDA language is used to launch many thousands, or even millions of threads to perform concurrent execution. In doing so, there are many design options and performance optimisations that can be applied to the CUDA code. This section explores the mapping of threads to the three-dimensional data, the use of shared memory, and also caching optimisations.

3.5.1 Overview of the CUDA program design

As the GPU device is physically separate from the CPU host, memory for the data grids is explicitly created on the device only. In order to minimise the transfer of data between the host and device, the output of the simulation is stored on the GPU at each iteration in time, and only transferred back to the host at the end of the simulation. Two kernels are used within the time loop. First, a kernel launches the threads to update the grid, and then a single thread kernel that updates the input and reads the output (see Appendix B.3 for the full code listing).

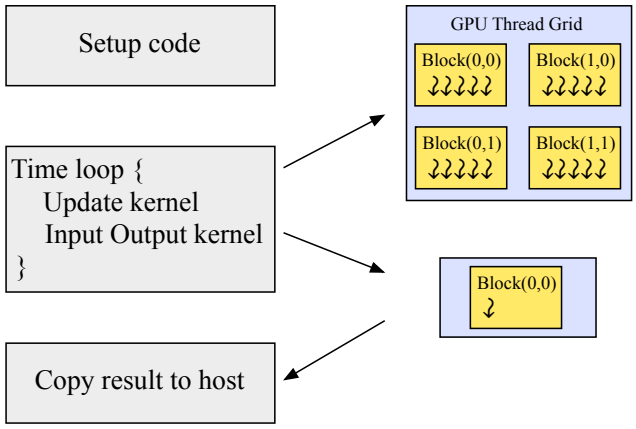


Figure 3.6 *Outline of the CUDA program design. The domain data is stored on the device only. The result array is copied back to the device after the time iteration has completed.*

3.5.2 Mapping threads to the data set

Recall that threads in CUDA are grouped together into a block, which can be one, two, or three-dimensional in terms of indexing. Thread blocks are set in a grid which can also have up to three dimensions. This architecture allows many possible arrangements for mapping threads over a three-dimensional data set. The first consideration in terms of the thread design is whether to issue enough threads to update the entire data set, or to issue threads that cover a two-dimensional slice only (Figure 3.7).

In the latter case, each thread then iterates over the remaining dimension to compute the updates. Whilst it is generally advantageous to maximise the number of threads being issued, the grid sizes being used here are very large. Every N_l by N_m slice of the data set contains 75,776 grid points, and this number of threads is easily enough to occupy the GPU device. The terms '3D tiling' and '2D slicing' are used to refer to each of these approaches.

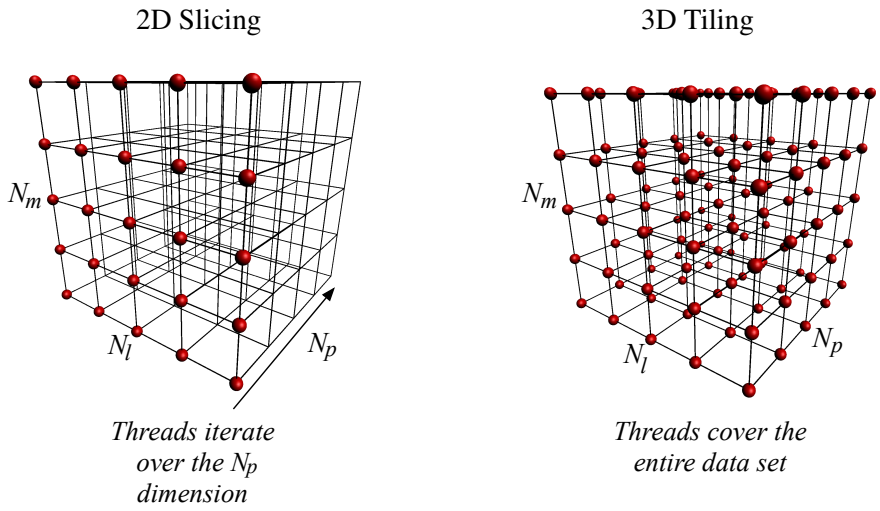


Figure 3.7 *Two approaches to threading the data. With 2D slicing each thread performs a loop over the N_p dimension. With 3D tiling enough threads are issued to cover the entire data grid.*

For the 3D tiling method, each thread simply obtains its three-dimensional indices and computes the update for its single grid point, as shown in the kernel at Algorithm 9. One conditional statement is required to ensure that the boundary nodes are not updated.

Algorithm 9 *3D tiling CUDA kernel for the basic 3D scheme.*

1: Get the 3D indices of the current thread from the block and thread IDs
2: **if** not at the boundaries **then**
3: Compute the linear address from the 3D indices
4: Update node $\Psi_{l,m,p}^{n+1}$
5: **end if**

The 2D slicing kernel differs in that each thread only obtains indices in the l and m dimensions. A `FOR` loop is then used to iterate over the N_p dimension, computing the updates for each grid point, as shown in Algorithm 10. In both approaches it is essential to ensure that consecutive threads access consecutive data elements from the linear memory, to obtain *coalesced* transfers. As the linear memory is decomposed using a row-major format, the first index of the thread ID (`threadIdx.x`) is used for the l index. In this simple test case simulation, the 3D grid is an integer multiple of the thread block sizes that are used, for example a $16 \times 16 \times 1$ block, or a $32 \times 4 \times 2$ block will cover the data without the need for memory padding to maintain coalesced transfers.

Algorithm 10 *2D slicing CUDA kernel for the basic 3D scheme.*

1: Get the indices in l and m of the current thread from the block and thread IDs
2: **if** not at the boundaries of l or m **then**
3: **for** $p = 1 : N_p - 2$ **do**
4: Compute the linear address from the 3D indices
5: Update node $\Psi_{l,m,p}^{n+1}$
6: **end for**
7: **end if**

3.5.3 Use of shared memory

Each block of threads has access to a fast local memory store known as *shared memory*. Making use of this memory space can reduce the amount of data that has to be read from the global memory, which is typically the performance 'bottleneck' in memory bandwidth limited algorithms. As the update equation for each grid point uses the six neighbouring points from the previous time step, each block of threads is accessing data that is used by multiple individual threads. By using a two-dimensional array

of shared memory, threads in a two-dimensional thread block can share data. Shared memory can be used for both the 3D tiling and 2D slicing methods. The latter method can also be combined with data reuse from register, as shown in Figure 3.8.

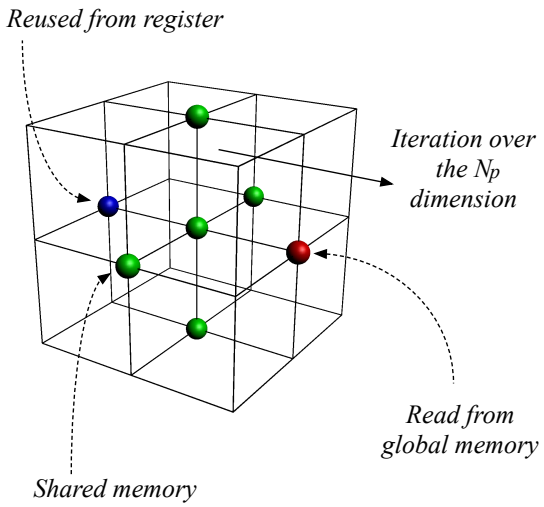


Figure 3.8 *Reducing global memory reads in the 2D slicing approach, using shared memory and register reuse.*

Using this method, each update calculation requires only one new read from global memory, rather than reading all six neighbouring points. This is a large reduction in the global memory requirements. However, the main complication for the 3D wave equation scheme is how to handle threads that are at the edges or corners of the 2D thread block, as shown in Figure 3.9.

There are two possible approaches here: using a standard size shared memory array or an extended size shared memory array. Using a standard size approach, the shared memory array is the same size as the thread block. In the example from Figure 3.9 this would be of size 32×8. The kernel then uses four conditional statements to test if a thread is at the boundary of the block, and if so to read the extra data directly from global memory.

The second approach is to use an extended shared memory array, and load the extra edge data that is required. Here the array would be of size 34×10. Conditional statements are used to direct threads at the edges of the block to load the extra data points. In the update, all data in the N_l and N_m dimensions are read from this extended shared memory array. See Appendices B.4 to B.7 for the full kernel codes.

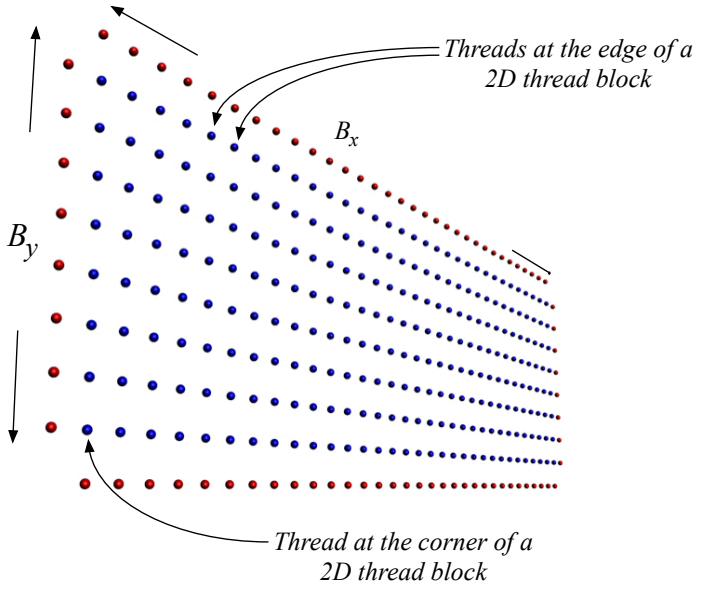


Figure 3.9 *A 32 x 8 thread block over the X and Y dimensions (in blue). Threads at the edges require one data point from outside of the block, threads at the corners require two. Data in the Z dimension is read from global memory in the 3D tiling method, or reused from register in the 2D slicing method.*

3.5.4 Cache optimisation

Aside from the design of the kernel code itself and experimenting with the thread block size, further efficiency can be obtained by optimising the use of the cache. Whilst the first generation Tesla GPUs had minimal cache levels, this has increased with each new generation of devices. On Fermi devices, using the `cudaFuncSetCacheConfig()` command to prefer the L1 cache can increase efficiency when only small amounts of shared memory are being used [100].

An additional feature of the Kepler architecture is the ability to use a read-only data cache which is separate from the standard L1 and L2 cache [90]. In the basic 3D wave equation scheme, the values from Ψ^n are only read from memory during any given time step (no writing is performed to that data grid). By declaring the pointer to that data in the kernel parameter using `const double * __restrict__ arrayname`, the data is read using this separate cache. In the following testing these cache optimisations were applied to the appropriate hardware for all kernels, and give efficiency gains between 10 to 15%.

3.5.5 Performance evaluation

The basic 3D scheme simulation was tested using two different Nvidia Tesla GPU devices: the Fermi architecture C2050, and the Kepler architecture K20 (see Appendix A.2 for the hardware specifications). Six different kernel designs are used to examine the differences between global memory usage and the two shared memory approaches. These are:

1. 3D tiling, reading from global memory.
2. 3D tiling with shared memory.
3. 3D tiling using the extended shared memory array.
4. 2D slicing, reading from global memory.
5. 2D slicing with shared memory.
6. 2D slicing using the extended shared memory array.

Each of the kernels is tested at single and double precision floating-point arithmetic, on both devices. The optimal thread block size was found to be $32 \times 4 \times 2$ for the 3D tiling method, and 32×8 for the 2D slicing method.

The results for the Fermi C2050 GPU device are shown in Table 3.3, for the Kepler K20 device in Table 3.4, showing both time and million grid points per second.

Kernel	**Single prec.** (min:sec)	**Mvox/s**	**Double prec.** (min:sec)	**Mvox/s**
3D tiling	2:01	5,744	3:49	3,035
3D tiling shared memory	2:44	4,238	5:53	1,969
3D tiling ext. shared memory	2:24	4,827	4:32	2,555
2D slicing	3:07	3,717	5:45	2,014
2D slicing shared memory	2:47	4,162	5:40	2,044
2D slicing ext. shared memory	2:07	5,473	3:45	3,089

Table 3.3 *Tesla C2050 GPU results for the basic 3D scheme at single and double precision. Mvox/s is million grid points per second.*

Kernel	**Single prec.** (min:sec)	**Mvox/s**	**Double prec.** (min:sec)	**Mvox/s**
3D tiling	1:24	8,275	2:35	4,484
3D tiling shared memory	2:07	5,473	3:35	3,233
3D tiling ext. shared memory	1:31	7,638	3:14	3,583
2D slicing	1:11	9,790	3:01	3,840
2D slicing shared memory	2:06	5,516	2:43	4,264
2D slicing ext. shared memory	1:29	7,810	2:26	4,761

Table 3.4 *Tesla K20 GPU results for the basic 3D scheme at single and double precision. Mvox/s is million grid points per second.*

The results vary depending on the device and the precision level. For the C2050 at single precision the 3D tiling approach is most efficient, whilst at double precision the 2D slicing with extended shared memory is optimal. On the K20, the basic 2D slicing is most efficient for single precision, but again at double precision the 2D slicing with extended shared memory gives the best result.

However, the basic 3D tiling approach is always very close to the most efficient solution, at both precision levels. The extra complications of the other approaches do not lead to large-scale improvements in the times, even with the reduction in global memory usage of the shared memory 2D slicing method. Comparing this to results from [139] using a consumer level GPU device, the differences between a basic implementation and a highly optimised version are much wider in that case. In terms of shared memory, the approach using the extended size array is always more efficient than the standard size. Table 3.5 shows a comparison of the speedups between the least and most efficient kernels on each device.

GPU device	**Speedup** (single prec.)	**Speedup** (double prec.)
Tesla C2050	1.6X	1.6X
Tesla K20	1.8X	1.5X

Table 3.5 *Speedups for the most efficient kernels over the least efficient kernels on each GPU device, for the basic 3D scheme.*

Finally, Figure 3.10 shows a combined comparison.

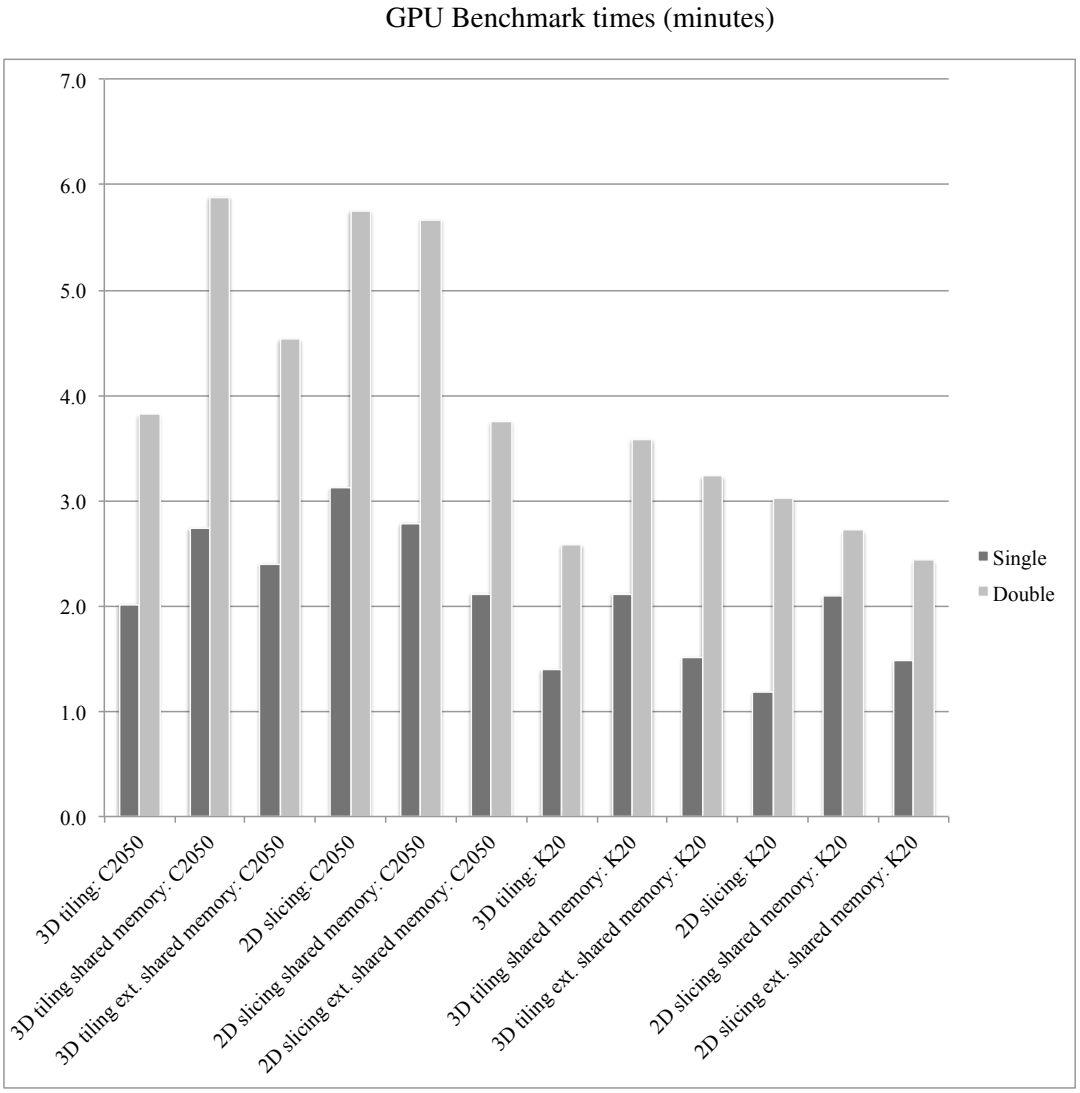


Figure 3.10 *Tesla GPU benchmarks for the basic 3D scheme at single and double precision.*

3.6 The use of multiple GPUs with CUDA

Section 3.5 demonstrated the extent of optimisation that is achievable on a single GPU device. In order to further reduce the computation times, multiple GPU devices can be used simultaneously to compute subsets of the data domain (in a similar manner to the pthreading approach for multi-core CPUs). Recent versions of the CUDA language allow multiple devices to be used without recourse to traditional MPI programming (Message Passing Interface) [140].

For scientific computing, Nvidia's Tesla devices are typically used in a workstation or compute node that can be configured with four GPUs connected across the same PCIe bus, as shown in Figure 3.11. Peer-to-peer communication allows data to be transferred between the devices that bypasses the host (normal transfers from host to device are very slow). This can be combined with the use of CUDA streams and asynchronous behaviour to achieve optimised performance.

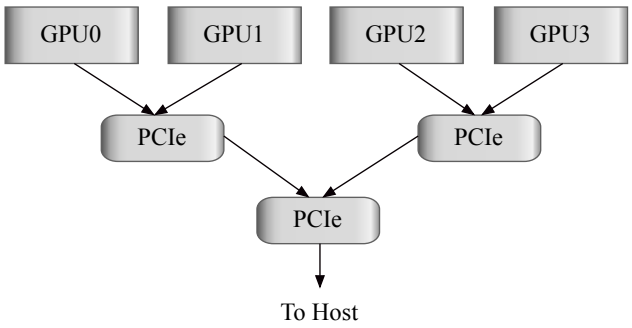


Figure 3.11 *Pair-wise connection of four GPU cards over the PCIe bus.*

Communication across devices has to be specified between the paired connections to the PCIe bus. In CUDA, this is performed by using a `cudaSetDevice()` command followed by enabling the peer access, as shown here between device zero and device one:

```
cudaSetDevice(gpu[0]);
cudaDeviceEnablePeerAccess(gpu[1], 0);
```

As the four devices have physically separate memory, the data domain has to be partitioned with specific *overlaps* to account for the data at the edges.

3.6.1 Data partitioning over multiple GPUs

Each of the two data grids used for the basic wave equation test is split into four segments, one for each GPU device. The partitioning is across the linear memory, in the same manner as for the ptheading, but with an extra N_p dimension layer held across the partition. These overlap 'halos' provide the N_p dimension data for the update equation. The outermost two devices need to store a single halo, whilst the innermost devices require two halos, as shown in Figure 3.12. The data in these halo layers must be transferred between the devices at each time step of the simulation, using data from the layer inside the partition line on the next device.

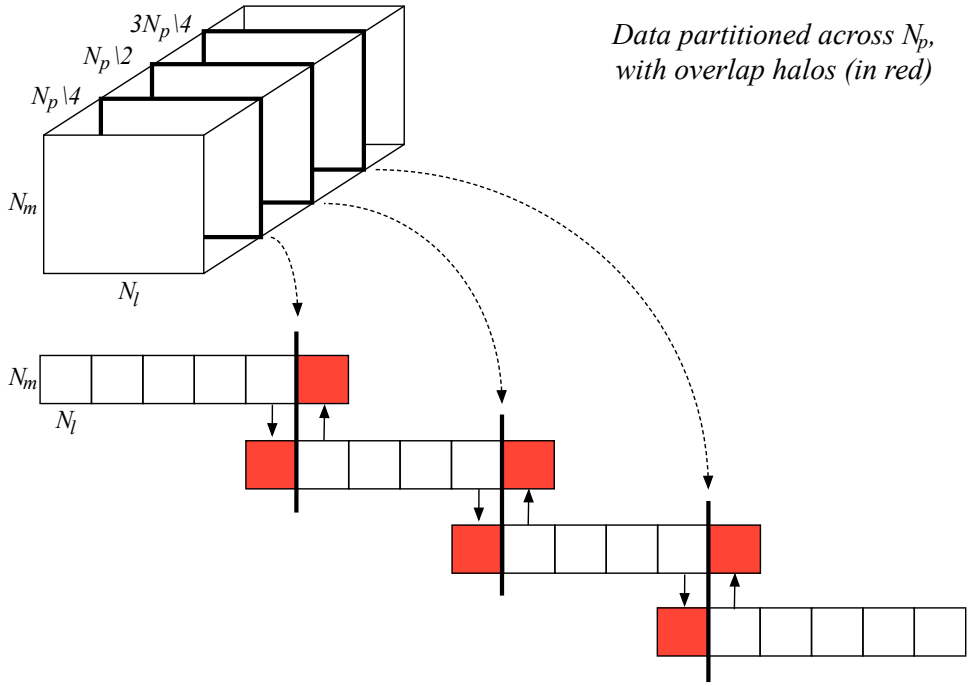


Figure 3.12 *Partitioning of the domain data over four GPU devices.*

There are two methods that can be used to implement the basic 3D wave equation simulation on four GPUs devices: one that uses non-asynchronous data transfers, and a more complex version using asynchronous transfers and streams.

3.6.2 Non-asynchronous implementation

The first implementation uses standard memory transfers that take place *after* the main data grid has been updated. The steps are outlined in Algorithm 11.

Algorithm 11 *Non-asynchronous implementation of the time iteration loop of the basic 3D scheme test simulation.*

1: **for** $n = 1 : NF$ **do**
2: Loop over the GPUs, issuing a kernel launch to compute all data on the device.
3: Synchronize all of the devices.
4: Perform peer-to-peer data transfers of the overlap halos.
5: Update the input and output.
6: Synchronize and then swap the data pointers.
7: **end for**

However, this approach contains an inherent time delay, as the devices are idle during the transfers of the halo data. In order to eliminate this, computation and data transfers need to occur simultaneously.

3.6.3 Asynchronous implementation

Two elements of CUDA functionality are required to implement an asynchronous approach: asynchronous peer-to-peer data transfers, and CUDA streams [110]. The first allows data transfers to occur asynchronously with respect to the host. The second, a stream, is simply a serial pipeline of events. However, by issuing kernel launches on different streams, the computation and data transfers can be executed at the same time (provided that there is no clash).

The data layers required for the halos can be computed first, and when they have completed the data transfers are started. At the same time, but on a different stream, the main non-halo data is computed, as described in Algorithm 12. The full CUDA code for the time loop iteration is listed in Appendix B.8.

Algorithm 12 *Asynchronous implementation of the time iteration loop of the basic 3D scheme test simulation.*

```
1: for n = 1 : NF do
2:     for n = 1 : numCards do
3:         Launch kernel to update the layers for the halo(s), using stream halo.
4:         Launch kernel to update the all of the remaining data, using stream interior.
5:     end for
6:     Launch asynchronous data transfer of halo data, using stream halo.
7:     Update the input and output.
8:     Synchronize and then swap the data pointers.
9: end for
```

The use of the same stream to launch the computation of the data for the halos, and then the data transfers themselves, guarantees that the updates are complete before the transfers begin.

3.6.4 Performance evaluation

Both the non-asynchronous and the asynchronous approaches were tested using the standard simulation for the 3D wave equation scheme. Four of the Tesla C2050 devices, and four Tesla K20 devices were employed. The timing results, as well of the relative speedup from the single card simulation time, are given in Tables 3.6 and 3.7. The times for the single device simulation use the basic 3D tiling method.

Version	**Single prec.** (min:sec)	**Speedup**	**Double prec.** (min:sec)	**Speedup**
Single device	2:01	-	3:49	-
Basic four device	0:58	2.1X	1:46	2.2X
Asynchronous four	0:35	3.5X	1:02	3.7X

Table 3.6 *Basic 3D scheme test simulation using four Tesla C2050 devices.*

The main point to note is that the results do not scale to 4X when using four devices, even in the most efficient approach. Whilst the C2050 device achieves a 3.7X speedup over the single device time at double precision, the K20 device only achieves a 2.5X speedup. Indeed, the actual times for both devices using the asynchronous approach

Version	**Single prec.** (min:sec)	**Speedup**	**Double prec.** (min:sec)	**Speedup**
Single device	1:24	-	2:35	-
Basic four device	0:47	1.8X	1:26	1.8X
Asynchronous four	0:35	2.4X	1:01	2.5X

Table 3.7 *Basic 3D scheme test simulation using four Tesla K20 devices.*

are practically the same. This indicates that the PCIe data transfer speed is limiting the efficiency of the K20 device, and a faster transfer speed would allow further increases in performance.

This can be verified by examining the efficiency for various different sizes of the N_p dimension layer, while keeping the overall volume of the data grid the same (i.e. by varying the size of another dimension as well). The data transfers for the halos each use one N_p dimension layer, of size N_l by N_m. The results for various sizes of layer using the K20 device and the asynchronous code are shown in Table 3.8. As the size of the N_p dimension layer is reduced, the computation times continue to fall rapidly. At double precision, the simulation now runs almost twice as efficiently for the smallest layer size. As a general optimisation principle for simulations of rooms, it is important to set the largest size dimension to N_p. This keeps the size of the halo layers to a minimum.

Layer size (grid points)	**Single precision** (min:sec)	**Double precision** (min:sec)
303,104	0:59	1:39
151,552	0:42	1:19
75,776	0:35	1:01
37,888	0:25	0.53
18,944	0:23	0:38
9,472	0:21	0:36

Table 3.8 *Variation in the size of the N_p dimension layer on the K20 device, for the basic 3D scheme.*

3.7 Summary

This chapter has demonstrated many possible approaches and optimisation strategies for computing a test case simulation using the basic 3D FDTD scheme. Table 3.9 gives a summary of the results from both the CPU benchmark times, and the GPU times, at single and double precision floating-point arithmetic.

Code version	**Single prec.** (min:sec)	**Double prec.** (min:sec)
Slowest CPU (Xeon -O0)	143:18	144:54
Fastest single thread CPU (i7 -O3)	21:01	23:52
Fastest multi-thread CPU (Xeon)	6:08	7:34
Fastest GPU, K20	1:11	2:26
Multiple GPU, 4 $\times$ K20	0:35	1:01

Table 3.9 *Summary of results for the basic 3D scheme test simulation.*

These timing results (excluding the multiple GPU) can be expressed as comparative speedup figures as shown in Table 3.10.

Code version	**Speedup** (single prec.)	**Speedup** (double prec.)
Fastest GPU over slowest CPU	121.1X	59.5X
Fastest GPU over fastest single thread CPU	17.8X	9.8X
Fastest GPU over fastest multi-thread CPU	5.2X	3.1X

Table 3.10 *Summary of GPU speedups for the basic 3D scheme test simulation.*

At single precision these range from 121X when comparing the slowest CPU result, to 5.2X when comparing the most efficient multi-threaded CPU version. At double precision, the fastest single GPU result is only 3.1X more efficient than the optimal CPU benchmark.

Whilst each of these results is a real, justifiable figure, it would seem clear that the 'fairest' and most realistic comparison is between the best benchmark time that is possible with a CPU to the best possible GPU time. In this case, the 5.2X and 3.1X

speedup figures are the headline result. This is somewhat surprising, given that many authors (including this one) have published papers showing comparisons between CPU and GPU benchmarks with speedups in the 80X to 100X range and beyond [141] [142] [143] [24]. These types of figures do not do justice to the compute capabilities of the CPU processor [144]. It should also be noted that the Intel Xeon CPU used for the benchmarks is by no means a high-end processor, and that more expensive Xeon CPUs could well achieve better performance, further reducing the speedups for the GPU times. For reference, the peak memory bandwidth of the Xeon processor is 42 GB/s, whilst that of the Tesla K20 is 208 GB/s (see Appendix A).

As a final comparison, Table 3.11 shows benchmark times for the basic 3D scheme computed over various grid volumes, ranging from $1m^3$ to $64m^3$. The CPU code uses the eight pthread version executed on the Xeon processor, and the GPU code is the fastest version for the K20 device.

Volume	**CPU (s)** (min:sec)	**GPU (s)** (min:sec)	**Speedup**	**CPU (d)** (min:sec)	**GPU (d)** (min:sec)	**Speedup**
$1m^3$	0:26	0:02	13.0X	0:26	0:03	7.9X
$2m^3$	0:41	0:04	11.7X	0:42	0:06	6.8X
$4m^3$	1:13	0:07	10.7X	1:15	0:12	6.3X
$8m^3$	1:30	0:14	6.7X	1:39	0:31	3.2X
$16m^3$	2:43	0:27	6.0X	3:11	1:02	3.1X
$32m^3$	6:10	1:12	5.2X	6:39	2:06	3.1X
$64m^3$	12:20	2:22	5.2X	13:20	4:20	3.1X

Table 3.11 *Variation in volume size, at single (s) and double (d) precision, for the basic 3D scheme.*

At $1m^3$, the GPU device takes just 2 seconds at single precision and 3 seconds at double. At this small volume level, the GPU code is significantly faster than the CPU code, by around 10X. As the volume levels increase, the difference between CPU and GPU results becomes less, tending to the range of 3X to 5X as reported above.

Chapter 4

Performance of alternative schemes

The basic scheme used in the previous chapter is only one of many different options for approximating the solutions to the 3D wave equation. It used a 7-point *stencil* consisting of the centre point and the six nearest neighbours as its approximation to the Laplacian operator. However, there are many different schemes that can be designed using various combinations of neighbouring points [145]. These do not have to be confined to regular cubic grids, as other mesh types are possible [146]. This chapter compares the computational efficiency of the following alternative schemes:

- The staggered grid formation of the basic scheme.
- The 27-point interpolated wideband scheme (IWB).
- A 13-point scheme on a face-centered cubic grid (FCC).

Whilst the staggered grid form is entirely equivalent to the basic scheme, the IWB and FCC schemes use different operators and grids, and so exhibit different dispersion characteristics [147]. This also means that assessing the efficiency between the schemes is more complex, as there are many possible comparisons that can be made. For example, one could compare the schemes that simulate the same physical volume of space, but this requires different volumes of grids as the spatial step varies between the schemes (when setting λ at the Courant limit in order to minimise dispersion in each case).

This chapter compares the performance of CUDA versions of the above alternative schemes to that of the basic 3D scheme from Section 2.1.4, running on the Tesla K20 device.

4.1 Benchmark simulation

The test case simulation computes 44,100 samples at 44.1kHz as before, but now three different volumes are considered that cover a wide range of physical grid sizes: 1m^3, 38m^3, and 250m^3. The grid sizes (number of data elements in each dimension of the 3D array), at the Courant limit where $\lambda = 1/\sqrt{3}$, are shown in Table 4.1. The spatial grid step is again 0.0135m. The 250m^3 simulation requires 0.82GB of memory for each data grid when using double precision floating-point.

Grid sizes	**Total grid points**	**Volume at 44.1kHz**
$64 \times 80 \times 80$	409,600	1m^3
$256 \times 296 \times 208$	15,761,408	38m^3
$640 \times 400 \times 400$	102,400,000	250m^3

Table 4.1 *Benchmark simulation dimensions.*

The benchmark performance of the basic 3D scheme running on the Tesla K20 GPU is shown in Table 4.2. This uses the 3D tiling approach with the read-only data cache optimisation from Section 3.5.5, as an equivalent approach is used for the CUDA versions of the alternative schemes.

Volume	**Single precision** (min:sec)	**Double precision** (min:sec)
1m^3	0:02	0:03
38m^3	1:24	2:35
250m^3	8:58	16:40

Table 4.2 *K20 GPU benchmarks for the basic 3D scheme.*

The relative performance of the subsequent schemes is calculated as a ratio given by the new time divided by the equivalent benchmark time from this table, and denoted as the *benchmark ratio*. This is 'speed-down' figure, i.e. by what factor are the alternative schemes slower than the benchmark for the basic 3D scheme.

4.2 Staggered grid formation

The basic 3D scheme is second order in time, but only requires a single data field at each of those time periods. This data represents the acoustic field, or a velocity potential. As described in Section 2.1.4 the wave equation can also be discretised into a coupled finite difference scheme in both pressure and velocity fields. The update scheme uses four separate data fields: v_x, v_y, and v_z are the three velocity fields, and P the pressure. The scheme is given as

$$(v_x)^{n+\frac{1}{2}}_{l+\frac{1}{2},m,p} = (v_x)^{n-\frac{1}{2}}_{l+\frac{1}{2},m,p} - \frac{T}{\rho X}\left(P^n_{l+1,m,p} - P^n_{l,m,p}\right) \tag{4.1}$$

$$(v_y)^{n+\frac{1}{2}}_{l,m+\frac{1}{2},p} = (v_y)^{n-\frac{1}{2}}_{l,m+\frac{1}{2},p} - \frac{T}{\rho X}\left(P^n_{l,m+1,p} - P^n_{l,m,p}\right)$$

$$(v_z)^{n+\frac{1}{2}}_{l,m,p+\frac{1}{2}} = (v_z)^{n-\frac{1}{2}}_{l,m,p+\frac{1}{2}} - \frac{T}{\rho X}\left(P^n_{l,m,p+1} - P^n_{l,m,p}\right)$$

$$\begin{aligned} P^{n+1}_{l,m,p} = P^n_{l,m,p} - \frac{\rho c^2 T}{X}\Big(&(v_x)^{n+\frac{1}{2}}_{l+\frac{1}{2},m,p} - (v_x)^{n+\frac{1}{2}}_{l-\frac{1}{2},m,p} + (v_y)^{n+\frac{1}{2}}_{l,m+\frac{1}{2},p} \\ &-(v_y)^{n+\frac{1}{2}}_{l,m-\frac{1}{2},p} + (v_z)^{n+\frac{1}{2}}_{l,m,p+\frac{1}{2}} - (v_z)^{n+\frac{1}{2}}_{l,m,p-\frac{1}{2}}\Big) \end{aligned} \tag{4.2}$$

This is known as the staggered grid formation, as both the spatial and temporal fields are offset by half grid points. As for the basic 3D scheme, a 'fixed' boundary condition was used for this initial testing, by setting the velocity elements to zero.

4.2.1 Implementation

The algorithm for computing the staggered grid scheme is slightly more complex than for the basic single field approach, as it requires a two stage approach. Firstly the three velocity fields are updated using the velocity components from the previous time step and the two neighbouring pressure values. Then the pressure field is updated, using these newly updated velocity values. Algorithm 13 details this process (compare this to Algorithm 6 for the basic 3D scheme).

As only the central point from the previous time step is used in each of the updates, the scheme can be implemented using just a single data grid for each of the four fields. Data is read from the grids, and then the updated values overwrite the previous data in memory (hence there are no pointer swaps at the end of the time iteration loop). In

terms of indexing the half grid points of the velocity fields, these are simply shifted up to align them with the integer indices used for the pressure grid.

Algorithm 13 *Staggered grid time loop iteration.*

for $n = 1 : NF$ **do** ▷ Loop over the time steps
  for $p = 0 : N_p - 1$ **do** ▷ Loop over the spatial dimensions
    for $m = 0 : N_m - 1$ **do**
      for $l = 0 : N_l - 1$ **do**
        if $l > 0$ **then**
          Update velocity node $(v_x)^{n+\frac{1}{2}}_{l,m,p}$
        end if
        if $m > 0$ **then**
          Update velocity node $(v_y)^{n+\frac{1}{2}}_{l,m,p}$
        end if
        if $p > 0$ **then**
          Update velocity node $(v_z)^{n+\frac{1}{2}}_{l,m,p}$
        end if
      end for
    end for
  end for
  for $p = 0 : N_p - 1$ **do** ▷ Loop over the spatial dimensions
    for $m = 0 : N_m - 1$ **do**
      for $l = 0 : N_l - 1$ **do**
        Update pressure node $P^{n+1}_{l,m,p}$
      end for
    end for
  end for
  Update the input and read the output.
end for

Note that the spatial grids are 'shifted' by half a step in order to match their indices to integer memory. For the parallel implementation using CUDA, two separate update kernels were used on a single stream. A kernel to perform the updates on all three of the velocity fields, followed by a kernel to update the pressure field alone. A 3D tiling method is used for the threading approach, issuing enough threads to cover the entire data grid in each case.

4.2.2 Performance evaluation

It is useful to compare the number of floating-point operations and the memory requirements per complete update for this staggered scheme as opposed to the basic second order approach.

- *Basic 3D scheme*: 7 data reads and 1 write, 7 floating-point operations.
- *Staggered grid scheme*: 14 data reads and 4 writes, 16 floating-point operations.

This assumes that the two update stages for the velocity and pressure are treated separately (i.e. no data is reused from registers). The performance of the staggered grid scheme on the Tesla K20 device is given in Table 4.3.

Volume	**Single prec.** (min:sec)	**Benchmark ratio**	**Double prec.** (min:sec)	**Benchmark ratio**
$1m^3$	0:07	3.5X	0:12	3.9X
$38m^3$	4:43	3.4X	8:53	3.4X
$250m^3$	30:20	3.4X	55:10	3.3X

Table 4.3 *Staggered grid simulation using the K20 GPU.*

The staggered scheme runs between 3.3X and 3.9X slower than the basic 3D scheme, in the worse case being for the $1m^3$ at double precision. Clearly the use of two kernel launches, along with the increase in the memory bandwidth requirements, has a considerable effect on the overall efficiency.

The staggered grid form also requires four data grids, compared to the two grids for the basic scheme, which halves the maximum physical space that can be simulated on a given device. In every respect the staggered grid scheme performs worse than the basic 3D scheme, to achieve the same simulation results. If specific velocity or pressure values are required, for instance to compute a particular boundary condition, these can be derived from the second order scheme.

4.3 Interpolated wideband and face-centred cubic

In practical use for audio simulations, the basic 3D scheme suffers from two issues: the spatial *cutoff frequency*, and *dispersion*. In order to get reliable results up to the Nyquist limit from an FDTD simulation requires the condition $\lambda \geq 1$ [145]. Therefore at the stability limit for the basic scheme there is a considerable reduction in the usable range of output frequencies. This is also compounded by numerical dispersion, which is maximised for propagation along the axis directions in each dimension. These deficiencies can be reduced by increasing the sample rate of the simulation, but at a large computational cost. For example, doubling the sample rate increases the computational load sixteen times, and requires eight times the amount of memory.

This then leads to the consideration of alternative schemes that can minimise the frequency cutoff and levels of dispersion for a given simulation [148]. This section compares the interpolated wideband (IWB) and face-centred cubic (FCC) schemes to the basic 3D scheme benchmarks. Three separate comparisons are made between these schemes, as each has different characteristics.

4.3.1 Description of schemes

The IWB scheme is one of a family of compact 3D explicit schemes. It is derived from the difference equation for compact explicit schemes [149], as given by

$$\begin{aligned} P^{n+1}_{l,m,p} = d_1(&P^n_{l+1,m,p} + P^n_{l-1,m,p} + P^n_{l,m+1,p} + P^n_{l,m-1,p} + P^n_{l,m,p+1} + P^n_{l,m,p-1}) \\ +d_2(&P^n_{l+1,m+1,p} + P^n_{l+1,m-1,p} + P^n_{l+1,m,p+1} + P^n_{l+1,m,p-1} + P^n_{l,m+1,p+1} + P^n_{l,m+1,p-1} \\ &+P^n_{l,m-1,p+1} + P^n_{l,m-1,p-1} + P^n_{l-1,m+1,p} + P^n_{l-1,m-1,p} + P^n_{l-1,m,p+1} + P^n_{l-1,m,p-1}) \\ &+d_3(P^n_{l+1,m+1,p+1} + P^n_{l+1,m-1,p+1} + P^n_{l+1,m+1,p-1} + P^n_{l+1,m-1,p-1} \\ &+P^n_{l-1,m+1,p+1} + P^n_{l-1,m-1,p+1} + P^n_{l-1,m+1,p-1} + P^n_{l-1,m-1,p-1}) \\ &+d_4 P^n_{l,m,p} - P^{n-1}_{l,m,p} \end{aligned} \tag{4.3}$$

where the parameters d_1, d_2, d_3, d_4 are defined as $d_1 = \lambda^2(1-4a+4b)$, $d_2 = \lambda^2(a-2b)$, $d_3 = \lambda^2 b$, and $d_4 = 2(1-3\lambda^2+6\lambda^2 a-4b\lambda^2)$. The IWB scheme is obtained by setting the free parameters $a = 1/4$ and $b = 1/16$, and setting $\lambda = 1$. The stencil uses the centre point and the 26 nearest neighbouring points, as shown in Figure 4.1.

This 27-point stencil has been shown to significantly reduce dispersion error [150], and has a stability limit of $\lambda \leq 1$. It is therefore capable of giving full bandwidth

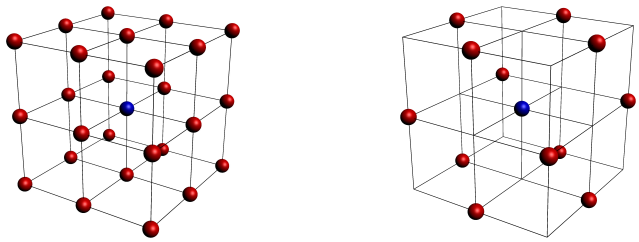


Figure 4.1 *Stencils for the 27-point IWB scheme (left) and 13-point scheme on an FCC lattice grid (right).*

output. The 13-point stencil also an approximation to the Laplacian derived from the generalized difference equation. A face-centred unit cell consists of a cubic cell with additional nodes located in the centre of each face. A lattice made of such cells is more efficient in its distribution of samples and use of space than a regular cubic lattice [151]. The FCC scheme examined here uses a 13-point stencil on a face-centred cubic grid, as shown in Figure 4.1. See [147] for a complete description. This lattice must be mapped to a regular cubic grid in order to implement the scheme in a simulation.

4.3.2 Implementation

The mapping of the FCC lattice to a regular cubic lattice is a two part process. Starting from a cubic lattice, one of the dimensions is scaled by two and then the alternate points in the remaining coordinates are shifted in the same direction (Figure 4.2).

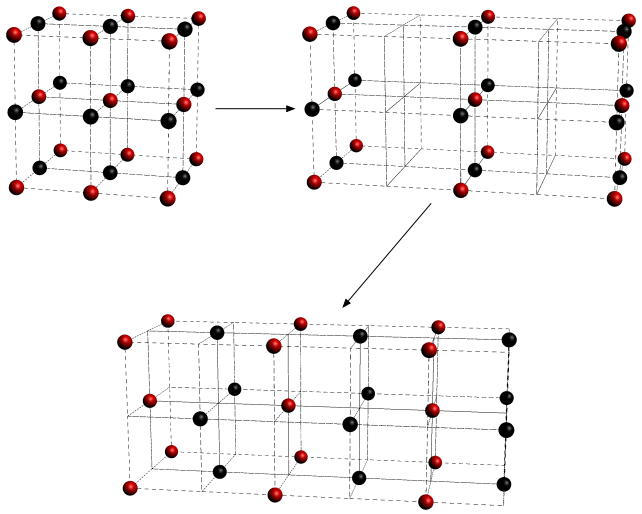


Figure 4.2 *Grid mapping for the FCC scheme. Starting from a cubic grid (top left), one dimension is expanded (top right), and then the red/black nodes are separated.*

The lattice is then scaled by $h/\sqrt{2}$ to normalise the length of the grid spacing. This is more efficient use of memory space than simply implementing the FCC lattice in a checkerboard manner using a single stencil orientation. The scheme then has a stability bound of $\lambda \leq 1/\sqrt{2}$. The update equation itself makes use of two different updates, one for even indexed points and one for odd indexed points through the regular grid, as shown in Figure 4.3.

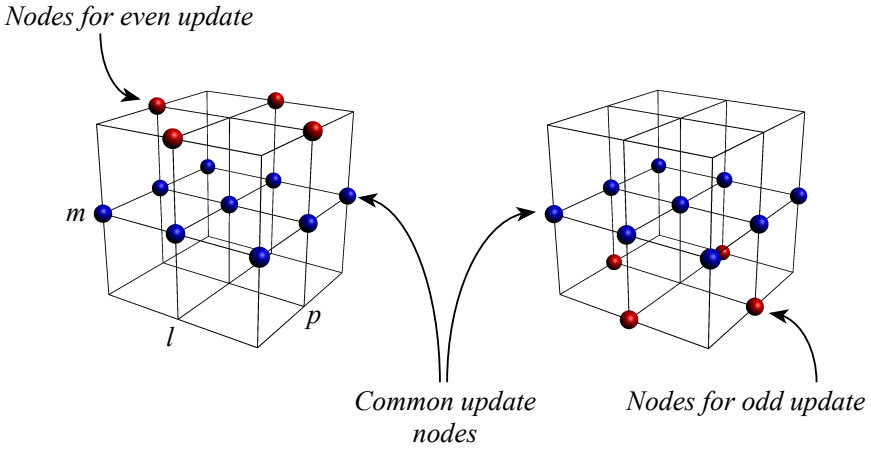


Figure 4.3 *Node points used for the FCC scheme odd and even node update equations.*

Implementing the updates in CUDA requires careful consideration to achieve full memory coalescing. Ideally the entire grid should be updated by a single kernel, as the use of multiple kernels will always be less efficient due to the overhead incurred in launching threads. In order to implement the dual updates in a single kernel and coalesce all of the transfers, the data for the nine common points as well as the data for both of the even and odd updates are loaded into register. Then, a conditional `IF -> ELSE` statement is used to load the required four data points into a temporary sum variable. See Appendix B.9 for the full kernel code.

The IWB scheme is implemented using the standard 3D tiling method, and both schemes use a 32×4 thread block. As per the basic 3D scheme benchmark, a fixed 'zero' boundary is employed in each case.

4.3.3 Performance evaluation

There are many comparisons that could be made between the IWB, FCC and basic 3D schemes. The optimal value of λ at the stability limit is different in each scheme, and the stencils differ in the density of the node points that are used. This performance evaluation makes the following three sets of comparisons, in each case testing the IWB and FCC schemes against the basic 3D scheme.

1. Test simulations using an equal number of grid points in each scheme, at the same sample rate.

2. Test simulations of equal physical size of space, at the same sample rate. Here the number of grid points varies.

3. Test simulations where the computational density of each scheme is normalised, giving a different sample rate and grid spacing in each case.

The first comparison is relevant from the perspective of running maximum memory tests on a given GPU device, where the simulation size is limited by the available global memory. The schemes are tested at the optimal λ values in each case, and for a set sample rate of 44.1kHz.

The second is useful to see the effect of running the different schemes at identical sample rates and physical simulation size, again using optimal λ values. The number of grid points in each scheme varies considerably in each case.

The third seeks to put each scheme on a level computational basis. The computational density is the number of grid points, or updates, per unit space and time. Fixing the Courant number at the stability bound and the wave speed across each scheme, the grid spacing (and hence sample rate) are varied to give an equal computational density across the schemes.

Each test case code is executed to produce a one second simulation output, for example at a sample rate of 44.1kHz then 44,100 time steps are computed. The codes are all performed on the Tesla K20 GPU device, at both single and double precision floating-point. The benchmarks for the basic scheme are those given in Table 4.2, except for the third comparison case where new benchmarks are computed.

Equal number of grid points

Tables 4.4 and 4.5 show the performance results when using an equal number of grid points as the benchmark basic 3D scheme. Recall that the physical sizes for that scheme were $1m^3$, $38m^3$ and $250m^3$. The sample rate for each simulation is 44.1kHz. Table 4.4 shows the results for the IWB scheme.

Volume	**Single prec.** (min:sec)	**Benchmark ratio**	**Double prec.** (min:sec)	**Benchmark ratio**
$0.2m^3$	0:04	2.1X	0:05	1.5X
$7.4m^3$	2:59	2.1X	3:14	1.3X
$48.3m^3$	20:05	2.2X	21:08	1.3X

Table 4.4 *IWB simulation using the K20 GPU and an equal number of grid points.*

Table 4.5 shows the results for the FCC scheme.

Volume	**Single prec.** (min:sec)	**Benchmark ratio**	**Double prec.** (min:sec)	**Benchmark ratio**
$0.4m^3$	0:03	1.5X	0:04	1.3X
$14.2m^3$	1:52	1.4X	3:01	1.2X
$94.7m^3$	12:46	1.4X	19:13	1.2X

Table 4.5 *FCC simulation using the K20 GPU and an equal number of grid points.*

Both the IWB and FCC schemes are relatively close to the benchmark at double precision, being only 1.3X and 1.2X slower. However, the extra data throughput leads to a worse time at single precision, the IWB being over twice as slow. Note the difference in the physical sizes being simulated here, the IWB scheme being less than one fifth the overall size of the basic 3D scheme.

Equal physical size

The second comparison uses simulations of the same physical size of space (and at the Courant limit of λ), requiring different grid sizes. For the IWB scheme, the 250m^3 simulation could only be performed at single precision due to the 5GB memory limit of the Tesla K20 device. The results for the IWB scheme are shown in Table 4.6.

Volume	**Single prec.** (min:sec)	**Benchmark ratio**	**Double prec.** (min:sec)	**Benchmark ratio**
1m^3	0:24	12.0X	0:25	8.3X
38m^3	15:55	11.4X	16:53	6.5X
250m^3	103:43	11.6X	-	-

Table 4.6 *IWB simulation using the K20 GPU and equal physical sizes of simulation compared to the benchmark.*

The IWB scheme is using five times as many grid points as the benchmark, and four times as many nodes points in each stencil, and yet still manages double precision times of 8.3X and 6.5X over the benchmark. The results for the FCC scheme are shown in Table 4.7.

Volume	**Single prec.** (min:sec)	**Benchmark ratio**	**Double prec.** (min:sec)	**Benchmark ratio**
1m^3	0:07	3.5X	0:09	3.0X
38m^3	5:06	3.6X	7:54	3.1X
250m^3	33:15	3.7X	50:57	3.1X

Table 4.7 *FCC simulation using the K20 GPU and equal physical sizes of simulation compared to the benchmark.*

Here the number of grid points is around 2.5 times the basic scheme benchmark, and each stencil is accessing twice the number of node points. In each of the comparisons, the FCC scheme results occupy a 'middle ground' between the basic 3D benchmark and the IWB scheme.

Equal computational density

For the final comparison the schemes are *normalised* across computational density, giving an equal number of updates, per unit space and time (see [147] for further details). Having adjusted the sample rate and grid spacing, each simulation is then computed at the three volume sizes from before, $1m^3$, $38m^3$ and $250m^3$. The new recomputed benchmark times for the basic 3D scheme with normalised density are shown in Table 4.8.

Grid dimensions (points)	**Single prec.** (min:sec)	**Double prec.** (min:sec)
64 x 68 x 64	0:01	0:02
256 x 200 x 194	0:41	1:24
480 x 400 x 338	5:06	9:17

Table 4.8 *Benchmark simulations using the K20 GPU for the basic 3D scheme with normalised computational density. The sample rate is 37.91kHz and grid spacing is 15.72mm.*

The results for the IWB scheme are shown in Table 4.9, and for the FCC scheme in Table 4.10.

Grid dimensions (points)	**Single prec.** (min:sec)	**Benchmark ratio**	**Double prec.** (min:sec)	**Benchmark ratio**
64 x 80 x 76	0:02	2.0X	0:03	1.5X
256 x 248 x 236	1:32	2.1X	2:13	1.6X
512 x 464 x 412	10:02	2.0X	14:52	1.6X

Table 4.9 *IWB simulations using the K20 GPU with normalised computational density. The sample rate is 25.11kHz and grid spacing is 13.70mm.*

Again, the IWB scheme suffers at single precision floating-point due to the amount of data access that is required for the 27-point stencil, running twice as slow as the benchmark standard scheme. The FCC scheme is considerably more efficient, especially at double precision where the runtimes are only 1.2X over the benchmark.

Grid dimensions (points)	**Single prec.** (min:sec)	**Benchmark ratio**	**Double prec.** (min:sec)	**Benchmark ratio**
64 x 80 x 80	0:02	1.8X	0:03	1.4X
256 x 240 x 216	1:14	1.8X	1:43	1.2X
512 x 412 x 400	7:55	1.7X	11:07	1.2X

Table 4.10 *FCC simulations using the K20 GPU with normalised computational density. The sample rate is 29.86kHz and grid spacing is 16.29mm.*

4.4 Summary

This chapter has examined the efficiency of GPU implementations for various alternative schemes for the 3D wave equation. The staggered grid formation with coupled velocity and pressure fields is shown to be 3.3X to 3.9X slower than the benchmarks for the basic 3D scheme. Three sets of comparison tests were performed using the 27 point IWB scheme and the 13 point FCC scheme against the basic 3D scheme. These used an equal number of grid points, then an equal physical size, and finally normalising across computational density.

The FCC scheme shows some efficiency gains over the IWB scheme in each case, and in terms of the first and third tests it produces timings close to the basic 3D scheme, running at 1.2X slower even though the scheme uses twice the number of grid points and has a more complex kernel arrangement.

The usefulness of these alternatives schemes is dependent on the level of dispersion that is acceptable for audio simulations, which also depends on the physical size of the simulation being produced. Detailed perceptual testing in this area is the subject of ongoing research.

Chapters 3 and 4 have detailed the use of GPU devices for accelerating simulations of three-dimensional wave propagation. However, in order to produce simulations of sound propagating in virtual spaces that approximate real world behaviour, several additional elements are required.

Chapter 5

Virtual acoustic simulations

One of the most important aspects of simulations in virtual acoustics is the boundary conditions that are used. This, along with the characteristics of the three-dimensional propagation, are the key factors in defining the quality of the resulting output. This chapter examines the implementation issues and the impact on efficiency of simple state-free boundary conditions, as well as those which require extra state data to be held in order to implement the boundary condition. It then details large-scale auralizations that use all available memory across four GPU devices, with data grids that contain billions of points. In order to maximise the limited physical memory available on a GPU device it is useful to run simulations using single precision floating-point arithmetic. Issues relating to the use of single precision are examined.

5.1 State-free boundary conditions

From an implementational perspective, state-free boundary conditions can be characterised by an update equation that only requires data that is already stored in the state grid from the previous time step. No additional state data needs to be held over time steps in order to compute the boundaries. This is of particular interest from an implementation perspective, as it facilitates the use of a single thread SIMD operation to update both the interior and boundary grid points. The simplest form of state-free boundary that is useful in terms of virtual acoustics is frequency-independent with an absorption term.

5.1.1 Frequency-independent lossy boundary

This simple resistive boundary is described as

$$\frac{\partial \Psi}{\partial t} = -c\beta \mathbf{n} \cdot \nabla \Psi \tag{5.1}$$

where $\mathbf{n}$ is an outward unit normal to the surface, and β here is an absorption coefficient. The stencil of the finite difference update scheme is adjusted for grid points where one or more legs of the stencil fall outside of the boundaries. Each of these stencil legs is folded back onto the centre point, as shown in Figure 5.1.

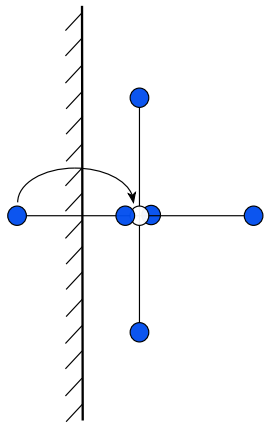


Figure 5.1 *Simple state-free boundary condition. Stencil legs that fall outside of the boundary are folded back onto the centre point.*

Recall the standard finite difference scheme for the 3D wave equation (2.14), which is

$$\Psi^{n+1}_{l,m,p} = (2 - 6\lambda^2)\Psi^{n}_{l,m,p} + \lambda^2 S - \Psi^{n-1}_{l,m,p} \tag{5.2}$$

where S is the sum of the six neighbouring points on the regular grid. To implement the frequency-independent lossy boundary, this becomes

$$\Psi^{n+1}_{l,m,p} = \frac{1}{1+\lambda\beta}\left((2 - K\lambda^2)\Psi^{n}_{l,m,p} + \lambda^2 S - (1-\lambda\beta)\Psi^{n-1}_{l,m,p}\right) \tag{5.3}$$

where K has a value of 5 at a face, 4 at an edge, and 3 at a corner, and S is the sum of only those neighbouring points that are inside the boundary, as detailed in [24].

5.1.2 Implementation methods

The implementation of this scheme requires two elements. At each type of boundary (face, edge, or corner) the correct stencil values must be obtained, and the coefficients of the update scheme need to be adjusted. The value of K is dependent on the type of boundary, and at all boundaries the coefficients containing the loss terms must be applied.

Ensuring that the correct stencil data is applied can be efficiently implemented using a "halo" layer around the simulation space. This is a single layer of data that is initialised to contain values of zero, and is never updated. Therefore, the standard stencil can be applied across both the interior and boundary positions without having to adjust the method of reading data from global memory. This is a single SIMD operation, and so maintains memory coalescing across both the interior and boundary grid points.

An optimal method for arranging the necessary coefficients using a single conditional statement and arithmetic that makes use of logical operators is shown in Algorithm 14. The value of K is computed using logical `OR` operators that reference the current thread indices. Then a single conditional statement sets the necessary loss coefficients.

Algorithm 14 *CUDA kernel for the frequency-independent lossy boundary using a single conditional statement.*

Get the 3D indices of the current thread from the block and thread IDs
if inside the halo layer **then**
    Compute the linear address from the 3D indices
    Set loss coefficients to default value of 1.0
    K = 0|(l-1) + 0|(l-(N_l-2)) + 0|(m-1) + 0|(m-(N_m-2)) + 0|(p-1) + 0|(p-(N_p-p))
    if K$<$6 **then**
        Set loss coefficients to lossy values
    end if
    Update node $\Psi^{n+1}_{l,m,p}$
end if

This is more efficient then using multiple conditional statements to apply separate updates for the interior, and then the face, edges, and corners. See Appendix B.10 for the full kernel code.

5.1.3 Performance evaluation

This frequency-independent lossy boundary scheme can be compared to the original lossless scheme for the 3D wave equation. A simple cube shape is used, with three different volumes as shown in Table 5.1.

Grid dimensions	**Total grid points**	**Volume**
$64 \times 64 \times 64$	262,144	0.6m^3
$256 \times 256 \times 256$	16,777,216	40.9m^3
$512 \times 512 \times 512$	134,217,728	329.1m^3

Table 5.1 *Benchmark simulation dimensions at 44.1kHz.*

The benchmark times for the basic 3D scheme running on the Tesla K20 are shown in Table 5.2. This uses the 3D tiling approach with the read-only data cache optimisation from Section 3.5.5.

Volume	**Single precision** (min:sec)	**Double precision** (min:sec)
0.6m^3	0:02	0:03
40.9m^3	1:30	2:45
329.1m^3	11:58	22:12

Table 5.2 *K20 GPU benchmarks for the basic 3D scheme with zero boundaries.*

The frequency-independent lossy boundary scheme was computed using a thread block of size $32 \times 4 \times 2$. The performance results are shown in Table 5.3.

Volume	**Single prec.** (min:sec)	**Benchmark ratio**	**Double prec.** (min:sec)	**Benchmark ratio**
0.6m^3	0:02	1.00X	0:03	1.00X
40.0m^3	1:33	1.03X	2:47	1.01X
329.1m^3	12:33	1.04X	22:28	1.01X

Table 5.3 *Frequency-independent lossy boundary simulation using K20 GPU.*

5.2 Boundaries requiring state memory

Whilst the frequency-independent lossy condition can create useable results, more realistic boundaries require conditions that are frequency dependent and have a complex resonant structures. The calculations for these types of condition require extra state data to be held at the boundary layers. This section addresses the impact of this form of calculation on overall efficiency, especially with consideration to memory coalescing on the GPU. A second order absorbing boundary condition is used to demonstrate this.

5.2.1 Engquist Majda boundary condition

The Engquist Majda condition is an absorbing boundary that produces minimal reflections from incoming waves [152]. This can be used in acoustics simulations to create an anechoic space for the purpose of analysis and testing, over a simple geometry like a cuboid. Whilst the usual choice in electromagnetics is the perfectly matched layer, the Engquist Majda condition is a simpler algorithm which is easier to implement. The conditions can be written as

$$\left(\frac{\partial}{\partial t} + c\mathbf{n} \cdot \nabla\right)^q \Psi = 0 \tag{5.4}$$

in terms of the order q.

Reflections from this boundary condition are confined to high frequency waves which are tangential to the boundary. Wavefronts are absorbed over an increasingly large range of frequencies and angles relative to the normal as q increases. As long the output is not read directly from the boundary layer then the conditions are perceptually transparent, even in the case of $q = 2$ which is used for testing here.

5.2.2 Implementation methods

The update equation applied to the interior grid points is the standard 3D wave equation form (2.14). However, we can no longer use a SIMD approach to update the boundary layers as well with a single kernel, as the method of updating the boundaries requires a more complicated system that references an extra layer of state data. This data has to be held in memory for both the current, previous, and second previous time steps. There is also a different update equation for the edges of the domain that does not use the extra layers of data.

The basic structure of the time iteration loop is to first update the interior grid points, followed by updating the six faces of the cube, and then finally the twelve edges. In terms of parallelisation for the GPU, a basic CUDA code would use the following algorithm, where DIM is the length of a side of the cubic domain space.

Algorithm 15 *Engquist Majda boundary time loop.*

1: Setup parameters
2: Create memory
3: **for** $n = 1 : NF$ **do** ▷ Loop over the time steps
4: Issue DIM^3 threads to update the interior grid points
5: Issue DIM^2 threads to update the six faces
6: Issue DIM threads to update the twelve edges
7: Update the input and read the output
8: Swap data pointers for state cubes and extra layers
9: **end for**
10: Process output and free memory

Every face update thread computes the value of one node on each of the six faces, and every edge thread computes a node on each of the twelve edges. The main issue in terms of performance relates to memory coalescing. Whichever way the data for the main grid is arranged in computer memory, there will always be two faces of this grid which do not contain contiguous data when reading across those faces (Figure 5.2). Referencing this data will necessarily be far slower than for the other four faces. The same applies to the updates for the edges. Only four of the twelve edges will be able to access data in a coalesced manner. Although edges only contain a small amount of data (of size DIM), the updates at the faces are substantial (DIM^2).

One method to minimise the inefficiency is to remap the main grid data required at the non-coalesced faces into new arrays that are arranged in a contiguous fashion. Whilst this may seem counter-productive (having to read and write the data), it is possible to mask this process by performing it inside the update for the interior grid points. Two layers of data are required, and remapped, at each non-contiguous face (Figure 5.3).

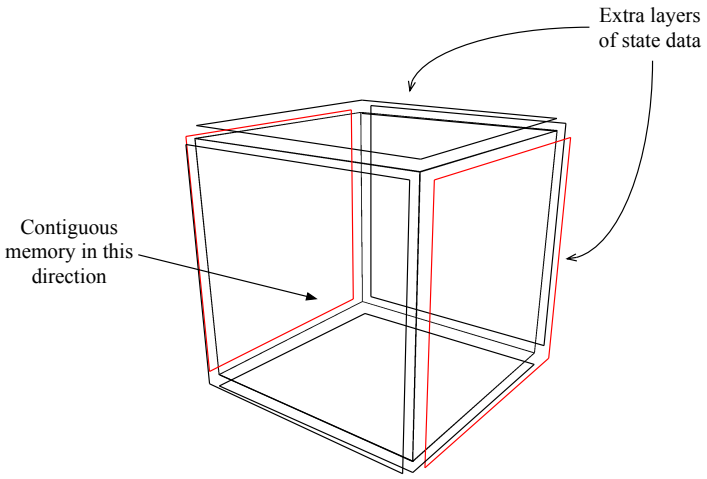


Figure 5.2 *An extra layer of state data is required at each face for the second order boundary condition. Faces in red have non-contiguous memory access when reading data from the main cubic data grid.*

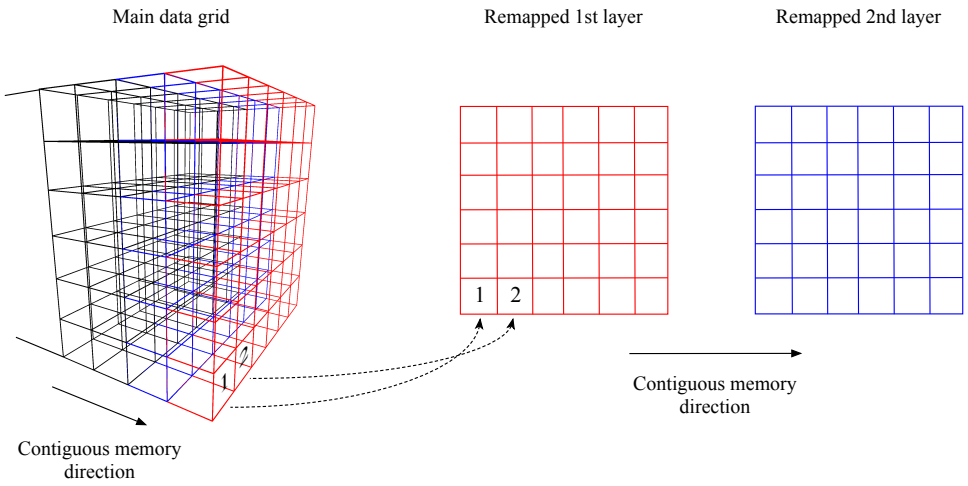


Figure 5.3 *Remapping the grid data for coalesced memory transfers.*

This is possible as threads at the outer boundaries of the interior update kernel are normally idle, as a conditional statement is used to check that the thread indices are inside the interior domain. By utilising some of these threads, the rearranging process can be partially masked whilst the bulk of the interior points are processed. See Appendix B.11 for the full CUDA code.

5.2.3 Performance evaluation

Both the basic method and the optimised version were tested at the simulation sizes given in Section 5.1.3, using the Tesla K20 device. The corresponding results are shown in Tables 5.4 and 5.5.

Volume	**Single prec.** (min:sec)	**Benchmark ratio**	**Double prec.** (min:sec)	**Benchmark ratio**
$0.6m^3$	0:05	2.5X	0:07	2.3X
$40.0m^3$	2:10	1.4X	3:35	1.3X
$329.1m^3$	15:39	1.3X	32:12	1.4X

Table 5.4 *Engquist Majda boundary simulation using K20 GPU*

Volume	**Single prec.** (min:sec)	**Benchmark ratio**	**Double prec.** (min:sec)	**Benchmark ratio**
$0.6m^3$	0:03	1.5X	0:05	1.7X
$40.0m^3$	1:54	1.3X	3:13	1.2X
$329.1m^3$	13:52	1.2X	24:05	1.1X

Table 5.5 *Optimised Engquist Majda boundary simulation using K20 GPU.*

Compared to the benchmarks for the basic 3D scheme, the initial method for the Engquist Majda boundary is up to 2.5X slower, with the worst case times at the smallest volume size. This performance decrease is entirely due to the extra threads updating the boundaries, as the interior is identical to the benchmark. Therefore it is taking more time to update the boundaries than to update the entire interior, at the smallest volume size.

The optimised method that uses the remapped faces is considerably more efficient. The benchmark ratios are now in the range of 1.5X in the worst case to just 1.1X at the largest volume size. This serves to highlight the importance of coalesced memory transfers in maximising efficiency.

5.3 Attenuation of sound in air

The wave equation derived in Section 2.1.1 is based on the assumption that sound waves propagate through air without losses. However in reality there are losses that occur, and these become evident in large-scale propagation spaces such as room acoustics simulations over several thousand cubic metres. As acoustic waves travel there is a frequency-dependent attenuation due to the viscous forces and thermal relaxation within the medium [13]. In order to simulate this, a viscosity element is added to the modelling equation.

5.3.1 Description of scheme with viscosity

The modified wave equation with viscosity is given by

$$\frac{\partial^2 \Psi}{\partial t^2} = c^2 \nabla^2 \Psi + c\alpha \nabla^2 \frac{\partial \Psi}{\partial t} \tag{5.5}$$

where the term with coefficient $c\alpha$ is the viscous damping component, derived through linearization of the Navier Stokes equations [10]. α is defined as

$$\alpha \approxeq \left(1.6(\gamma - 1) + 4/3 + \frac{\eta}{\mu}\right) l/\sqrt{\gamma} \tag{5.6}$$

where l is the mean free path of molecules in air, γ is the ratio of specific heats, and η and μ are coefficients of viscosity. When α is small this leads to a frequency-dependent damping.

The modified finite difference scheme for the interior grid points is given by

$$\begin{aligned} \Psi^{n+1}_{l,m,p} = {} & (2 - 6\lambda^2)\Psi^n_{l,m,p} + \lambda^2 S - \Psi^{n-1}_{l,m,p} \\ & + \alpha c T \left((S - 6\Psi^n_{l,m,p}) - (S' - 6\Psi^{n-1}_{l,m,p})\right) \end{aligned} \tag{5.7}$$

where S' is the sum of the six neighbouring points in $\Psi^{n-1}_{l,m,p}$. The frequency-independent lossy boundary condition can be used as per Section 5.1.1. The stability condition for the scheme follows from von Neumann analysis, and the grid spacing must satisfy the following

$$X \geq \sqrt{3c^2T^2 + 6\alpha cT} \tag{5.8}$$

which is slightly larger than for the standard wave equation.

5.3.2 Performance evaluation

As neighbouring data from $\Psi_{l,m,p}^{n-1}$ is required for the update, this means that three different data arrays must be stored in memory instead of two for the basic scheme. Therefore 50% more memory is required for the same physical size of simulation. A practical value of α is around 2×10^{-7}. The scheme was tested at the simulation sizes given in Section 5.1.3 using the Tesla K20 device, and a $32 \times 4 \times 2$ thread block. Table 5.6 shows the results.

Volume	**Single prec.** (min:sec)	**Benchmark ratio**	**Double prec.** (min:sec)	**Benchmark ratio**
0.6m^3	0:03	1.5X	0:04	1.3X
40.0m^3	2:20	1.5X	4:04	1.5X
329.1m^3	18:03	1.5X	32:18	1.5X

Table 5.6 *Performance of the viscosity scheme using the K20 GPU.*

The extra data reads from global memory lead to a decrease in performance when compared to the benchmark scheme, being around 1.5X slower. However, the resulting audio output is significantly improved by the inclusion of the viscosity element, especially at audio rates such 44.1kHz. At this sample rate, the cut off frequency and levels of dispersion mean that there is minimal useable output above 10 kHz. The viscosity element 'smooths' out these high frequencies in large-scale room simulations, and reduces the buildup of energy there. Various audio examples of simulations that use this viscosity scheme are available on the accompanying DVD media. The examples use dry audio files that are directly input into the scheme during the runtime as a soft source.

5.4 Large-scale models

Previous sections have examined the efficiency of schemes using up to a hundred million points in each data grid. To produce realistic simulations of large-scale virtual acoustics at sample rates such as 44.1kHz requires an order of magnitude increase in the size. It is possible to create grids containing over a billion points by using all of the available memory across multiple GPU devices, and this translates to physical sizes of several thousand cubic metres.

This section examines the use of single precision floating-point arithmetic to achieve maximum memory usage, and the effect of dispersion in the large-scale simulation spaces that can be computed. Whilst the typical use of such simulations is to create a static impulse response, this is not the only possibility [153]. Audio sources can be injected into the virtual space in a dynamic manner, and output can be taken from the acoustic field at any position over the runtime of the simulation.

5.4.1 Maximum GPU memory usage

The following table shows the maximum grid sizes and physical simulation space at 44.1kHz and at the Courant limit for various GPU devices. This assumes single precision, using the basic two grid (Chapter 3) and three grid viscosity schemes.

GPU size	**2 grid scheme** (million points)	**Volume** (m^3)	**3 grid scheme** (million points)	**Volume** (m^3)
3Gb	352	865	235	584
5Gb	595	1,461	395	983
6Gb	722	1,774	480	1,193
4 x 3Gb	1,409	3,460	940	2,336
4 x 5Gb	2,430	5,960	1,582	3,934
4 x 6Gb	2,889	7,096	1,920	4,775

Table 5.7 *Maximum simulation sizes in points per grid and cubic metres, at 44.1kHz.*

For example, using four of the Tesla K20 devices the maximum simulation size is 5,960m^3 using a two grid scheme. This could simulate a space of size 26m by 16m by 14m, which is the size of a small concert hall venue [154]. The simulation time is 54 minutes for 44,100 time steps (i.e. a second of output), using the asynchronous multi-card method from Section 3.6.3.

5.4.2 Single precision issues

The use of single precision floating-point data objects is essential for maximising the physical size of a simulation for a given GPU system, where there is a finite amount of global memory. However, this does create two issues. These are the instabilities that can arise when using even the simplest of boundary conditions [155], and the accuracy of the data field when compared to double precision computation.

Instability

Whilst a fixed zero boundary condition is always stable, it is not actually useful for producing acoustic simulations. Testing of the frequency-independent lossy boundary condition from Section 5.1.1 at single precision on a 100 million point domain reveals stability issues when running at, or very close to, the Courant limit. This domain size was tested using the basic non-viscosity scheme. Audio input is injected as a soft source, using a short one second, dry recording of a guitar strum (DC-blocked).

The output is read from a single grid point approximately one metre away from the source. Figure 5.4 shows the outputs at single and double precision for a 40,000 time step simulation at 44.1kHz and the grid spacing set exactly at the Courant limit.

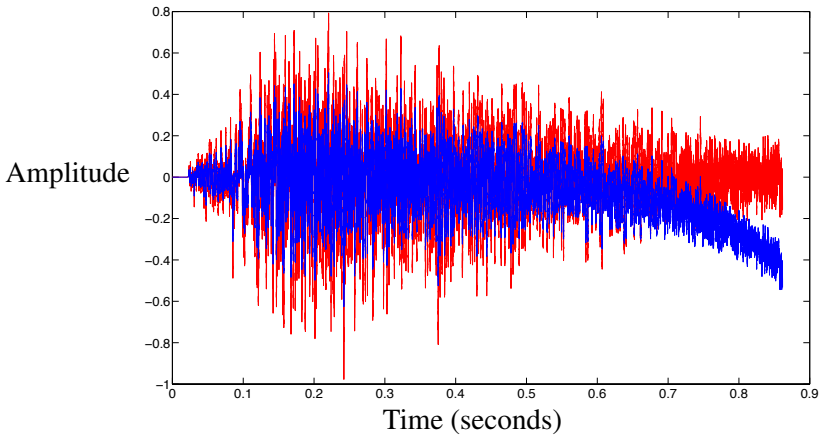


Figure 5.4 *Double (red) vs single (blue) precision simulations at the Courant limit.*

The single precision output (blue) is stable initially, but shows phase and amplitude differences compared to the double precision (red). After 30,000 samples the single precision output begins to diverge, and finally becomes unstable after 40,000 samples where it begins to exponentially drift to infinity (in this case the bound of floating-point representation) . Backing away from the Courant limit by around 0.4% ensures stability in single precision, but at the cost of introducing slightly greater dispersion.

Accuracy

The previous simulation was then repeated for both double and single precision with the grid spacing increased by 0.4% in both cases to test the accuracy of the resulting acoustic fields. Figure 5.5 shows the output signal of the double precision simulation, a guitar strum with a one second reverb tail.

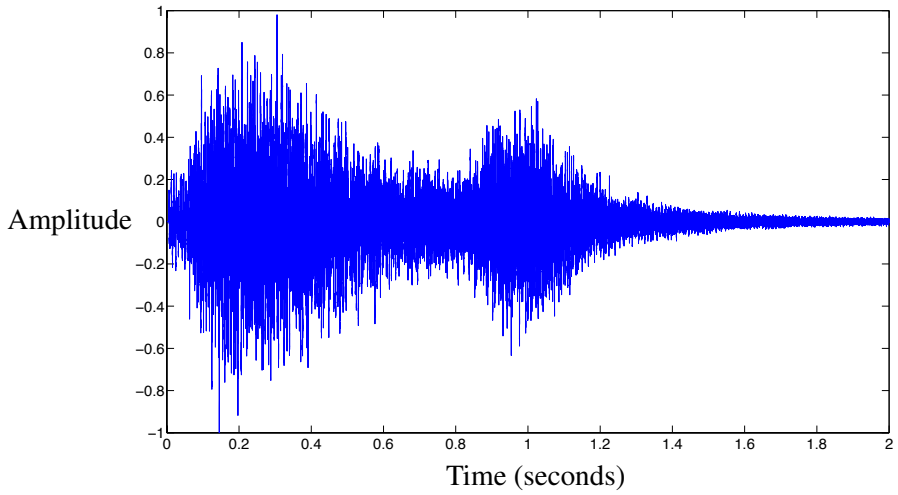


Figure 5.5 *Double precision output signal, over a two second simulation.*

The differences between the outputs of the single and double precision simulations at each time step are shown in Figure 5.6. The signals are both normalised by the maximum value of the double precision output, before calculating the differences.

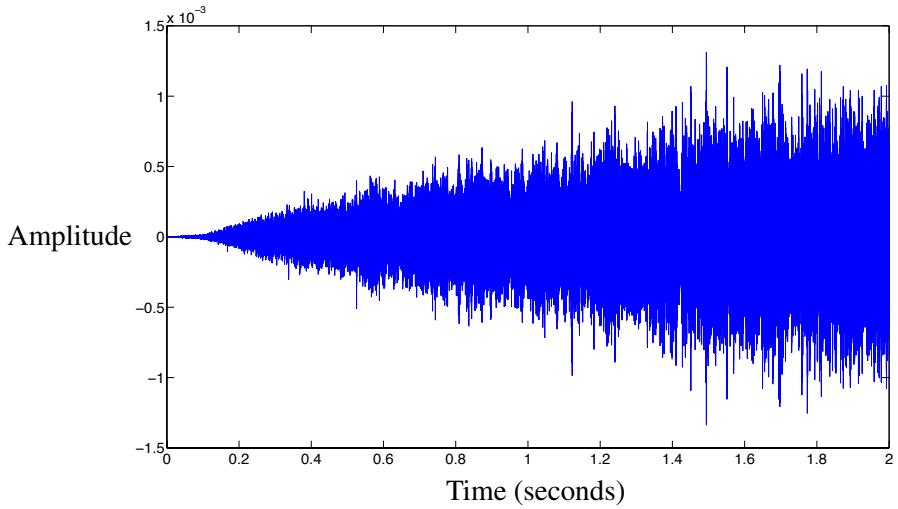


Figure 5.6 *Differences between normalised outputs of double and single precision.*

Machine accuracy ϵ_m is defined as the smallest floating-point number that produces a difference when added to the floating-point value of 1.0. The IEEE 754 standard single precision `float` has ϵ_m of around 1.2×10^{-7}, compared to 2.2×10^{-16} for 64 bit double precision [119]. At the beginning of the simulation the differences are in the order of 10^{-7} as expected. However, differences soon develop in the order of 10^{-3} which is significantly more than the base level accuracy. The perceptual effect of this loss of accuracy requires further testing.

5.4.3 Dispersion in the standard scheme

Using the maximum available memory over four Tesla K20 devices allows for simulations at 44.1kHz that approach the sizes of small concert halls. However the usability of such simulations is impeded by the levels of dispersion that occur in such large-scale simulations.

This can be demonstrated using a simple cubic space computed at a sample rate of 44.1kHz. The basic 3D scheme is used, along with a fixed zero boundary. A twenty sample raised cosine is used as an input to the scheme, summed in at a grid point in the centre of the cube. The simulation is terminated before the impulse reaches the boundary of the cube.

Figure 5.7 show the results from readouts taken on a straight line from the source position along the X dimension grid axis. Even at a distance of 1m there are additional oscillations after the initial impulse, and at 5m the impulse has significantly broken down into a series of oscillations.

Compare this to the output taken on a diagonal of the three grid axes where there is minimal dispersion, as shown in Figure 5.8. The raised cosine impulse remains unchanged, even at a distance of 5m away from the source.

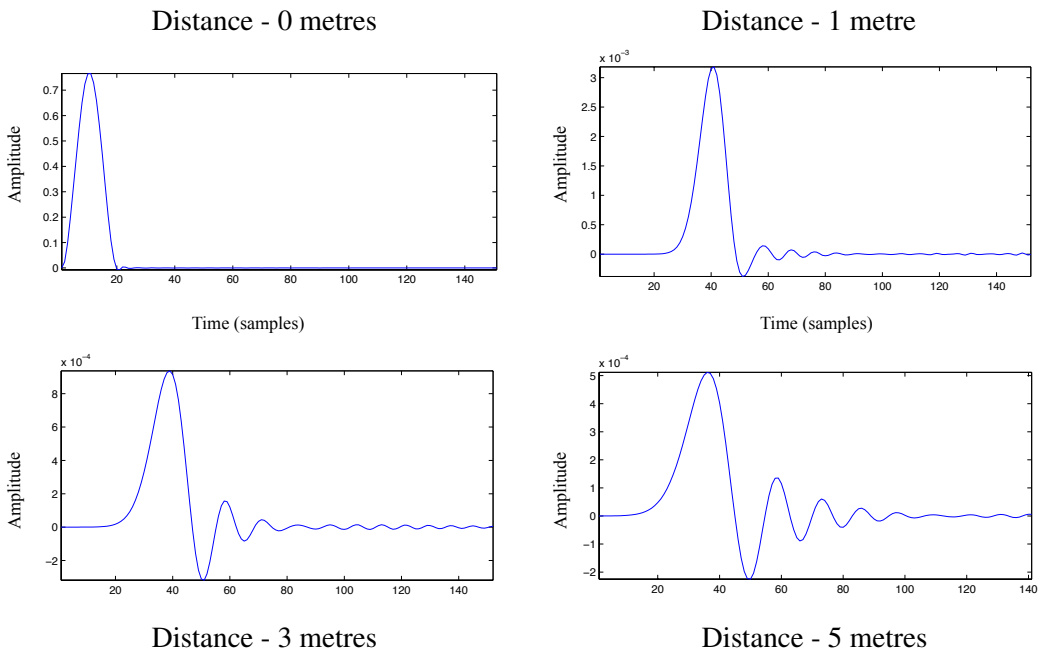


Figure 5.7 *A raised cosine impulse after travelling an increasing distance away from the source, with the read out performed along a grid axis. Each graph plots amplitude over a time frame of 150 samples.*

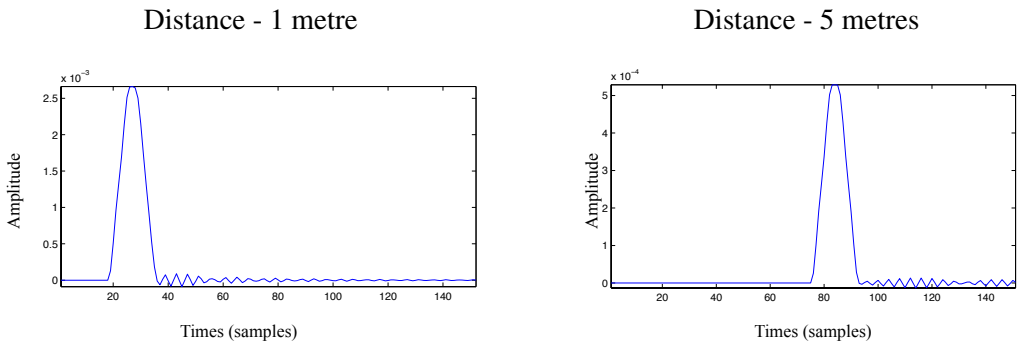


Figure 5.8 *A raised cosine impulse after 1m and 5m away from the source, with the read out performed on a diagonal of the three grid axes. Each graph plots amplitude over a time frame of 150 samples.*

5.5 Summary

The efficient implementation of boundary conditions is a critical factor for virtual acoustic simulations, particularly in the case of memory coalescing on the GPU. A SIMD operation that can update both interior grid points as well as the boundary, without the use of multiple conditional statements, is an optimal strategy, as shown by the simple frequency-indepedent lossy boundary tested in this chapter. For more complex boundary conditions that require extra state to be stored, an efficient solution of remapping the two non-contiguous faces has been demonstrated here.

The efficiency of a more advanced scheme that includes a viscosity element was tested, and this has useful properties for improving the quality of the output of a simulation. This scheme does make use of three state data grids, and so requires 50% more memory than the basic 3D scheme. In order to maximise simulation sizes on the GPU, single precision floating-point computation is necessary. However, this can lead to issues relating to instability and accuracy of the output, as has been demonstrated.

The ability to create very large-scale simulations allows the examination of dispersion characteristics in the standard scheme. The method of inputing the source has not been explored, as the main purpose here is for embedded physical models. See [156] for excitation methods relating to impulse responses.

Part III

Integrated physical models of instruments

Chapter 6

Basic linear algebra operations

The previous chapters of this thesis focused on three-dimensional wave propagation in free space, with a boundary domain termination. The following chapters move a step further and consider the performance issues involved in creating models of acoustic instruments that are actually *embedded* inside these spaces. Both one-dimensional systems such as strings, and two-dimensional systems such as membranes can be coupled into the three-dimensional space. Simple boundary conditions can also be used to create shells and other fixed objects that the vibrating systems are attached to. In this manner, a complete model of a complex instrument such as a drum can be simulated with the vibrations of the membrane propagating out into the three-dimensional space that surrounds it. Given the sizes of the simulations space achievable with even a single GPU device, it is then possible to create a simulation where one or more instruments can be placed within an actual room or hall model and computed at an audio sample rate such 44.1kHz.

As detailed in Section 2.4, the finite difference schemes for systems such as membranes can be represented in a generalised vector matrix form as

$$\mathbf{A}\mathbf{u}^{n+1} = \mathbf{B}\mathbf{u}^{n} + \mathbf{C}\mathbf{u}^{n-1} \tag{6.1}$$

where $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ are update matrices. For explicit schemes the matrix $\mathbf{A}$ is the identity, but for implicit schemes this is not the case and solving for $\mathbf{u}^{n+1}$ requires a linear system solution at each time step of the simulation.

Implicit methods are often required for stability in certain cases, and are also used to reduce dispersion. If the scheme is implicit and nonlinear the matrix $\mathbf{A}$ may be dependent on previously computed values of the solution $\mathbf{u}^{n}$, and will need to be constructed at each iteration before performing the linear system solution. The perfor-

mance of basic linear algebra operations is therefore important when creating simulations using these types of embedded systems. As the matrices involved are typically large and banded in structure they are very sparse, containing only a small percentage of non-zero values. Storing and processing such matrices in a normal dense array format would be extremely inefficient, and thus sparse matrix storage formats need to be employed along with functions for performing basic linear algebra on these formats. This chapter considers both generalised and optimised storage formats, and details the performance of basic operations on CPU and GPU hardware.

6.1 Sparse matrix storage formats

Sparse matrix storage seeks to hold only those non-zero entries from a matrix rather than the full version which may contain many unused zero entries. Consider the following matrix which contains values only along three of its diagonal bands.

$$\mathbf{A} = \begin{bmatrix} 5.5 & 0 & 4.2 & 0 \\ 4.1 & 3.9 & 0 & 1.9 \\ 0 & 2.7 & 4.0 & 0 \\ 0 & 0 & 1.4 & 6.9 \end{bmatrix}$$

There are many different storage formats that can be used for such a matrix, which vary from generalised forms to those which are optimised for such structures. Four formats are detailed here: CSR, CSC, DIAgonal, and ELLPACK.

6.1.1 CSR and CSC formats

The Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) formats are generalised systems that are suitable for any structure of matrix [119]. Both require three arrays of data; one to hold the real floating-point values of the entries, and then two integers arrays that contain positional data. The matrix **A** in CSC format is

```
int i [ ]  = { 0,   1,   1,   2,   0,   2,   3,   1,   3   }
int p [ ]  = { 0,        2,        4,             7,        9 }
real x [ ] = { 5.5, 4.1, 3.9, 2.7, 4.2, 4.0, 1.4, 1.9, 6.9 }
```

Here the array `i` holds the row of each entry, but the array `p` holds only the enumerated values that indicate the first element of each column (the index position in array `x` of the first element for each column). This array contains $n + 1$ entries by convention, where n is the number of columns in $\mathbf{A}$. The other arrays are of size nnz($\mathbf{A}$), the number of non-zero entries.

The matrix $\mathbf{A}$ in corresponding CSR format is

```
int p [ ]  = { 0,        2,              5,        7,          9 }
int j [ ]  = { 0,   2,   0,   1,   3,   1,   2,   2,   3   }
real x [ ] = { 5.5, 4.2, 4.1, 3.9, 1.9, 2.7, 4.0, 1.4, 6.9 }
```

Here the array `j` holds the column of each entry, and the array `p` holds the compressed row entries enumerating the first value in each row. Whilst both the CSC and CSR formats save memory space by compressing one of their integer arrays, there are other more optimised formats that can further reduce memory usage by utilising the banded structure of the matrices.

6.1.2 DIAgonal format

The diagonal, or DIA, format is specifically designed to use minimal memory space for diagonally banded matrices. Each band is mapped onto a single vertical column of a 2D table, and an integer array indicates the position of each band away from the centre diagonal. The matrix $\mathbf{A}$ in DIA format becomes

$$\mathbf{A}_{map} = \begin{bmatrix} -1 & 0 & 2 \end{bmatrix}$$

$$\mathbf{A}_{DIA} = \begin{bmatrix} . & 5.5 & 4.2 \\ 4.1 & 3.9 & 1.9 \\ 2.7 & 4.0 & . \\ 1.4 & 6.9 & . \end{bmatrix}$$

In terms of actual memory arrays this is stored as

```
int map [ ]  = { -1, 0, 2 }
real x [ ] = {0, 4.1, 2.7, 1.4, 5.5, 3.9, 4.0, 6.9, 4.2, 1.9, 0, 0}
```

The data can be addressed in a column-major format using C×rows+R, where R and C are the row and column location in the table. The use of column-major decomposition is an optimisation discussed in Section 6.4.1. Note that the DIA format does include some redundant data space, at the beginning of bands under the centre diagonal, and at the end of bands above the centre diagonal. For very large matrices with bands that are some distance apart this can be a significant amount of data. However, there is still a considerable reduction in memory usage compared to the CSR or CSC formats with their long integer arrays.

6.1.3 ELLPACK format

The DIA format is only optimal for matrices with strict diagonal bands. If any data falls outside of this structure, then the format quickly becomes inefficient. For matrices that contain data on the majority of rows, and a roughly consistent number of elements on each row, then the ELLPACK format presents a more optimal solution. Here, a initial 2D table contains the elements of each row, with the centre diagonal placed at the first column by convention. Then a second table contains the actual integer column indices for each entry in the first table. The matrix **A** becomes

$$\mathbf{A}_{ELL} = \begin{bmatrix} 5.5 & 4.2 & . \\ 3.9 & 4.1 & 1.9 \\ 4.0 & 2.7 & . \\ 6.9 & 1.4 & . \end{bmatrix} \mathbf{A}_c = \begin{bmatrix} 0 & 2 & . \\ 1 & 0 & 3 \\ 2 & 1 & . \\ 3 & 2 & . \end{bmatrix}$$

This is stored in linear arrays as

```
int c  [ ] = {0,   1,   2,   3,   2,   0,   1,   2,  -1, 3,  -1,-1}
real x [ ] = {5.5, 3.9, 4.0, 6.9, 4.2, 4.1, 2.7, 1.4, 0, 1.9, 0, 0}
```

with both tables decomposed in a column-major format. This gives a more flexible system for matrices that are not strictly banded, whilst still requiring less memory usage than the CSR or CSC formats.

6.2 Vector addition

Before examining the performance of matrix operations using the above formats, this section begins by detailing the most basic operations that can be performed on vectors, namely addition and the dot product. These operations feature heavily in the computation of finite difference schemes for embedded systems, and so it is useful to assess the performance of such operations on both CPU and GPU hardware, including benchmarking the CPU for multi-threaded performance.

6.2.1 CPU performance evaluation

Simple vector addition was tested for three different sizes of vectors; 10,000 elements, 100,000 elements, and finally one million elements. The operations were performed in a loop over 44,100 iterations to mimic their usage in a typical finite difference time domain simulation. Table 6.1 shows the results for single and multi-threaded code on both the Intel i7 and Xeon processors. The codes are compiled with -O3 optimisation. Timing results for the multi-threaded versions are given in seconds only, as CPU clock timers do not give accurate results when using threads over multiple cores.

Process	**size: 10,000** (seconds)	**size: 100,000** (seconds)	**size: 1,000,000** (seconds)
i7 - 1 thread	**0.3**	**4**	72
i7 - 4 threads	3	**4**	69
i7 - 8 threads	4	5	69
Xeon - 1 thread	0.4	5	90
Xeon - 6 threads	9	11	36
Xeon - 12 threads	15	20	**35**

Table 6.1 *Vector addition over 44,100 iterations, double precision floating-point.*

The multi-thread code is using a standard pthread process, with each thread computing results for a section of the vector. Up to a size of 100,000 elements the single thread version performs faster than the multi-threaded, sometimes considerably so. At small sizes the latency involved in issuing multiple threads and synchronising at each time step leads to much slower performance than a simple single thread code.

Only when the size is increased to one million elements does the multi-thread code achieve significant efficiency, and then only on the Xeon processor. The timings given in Table 6.1 are for double precision floating-point.

6.2.2 GPU performance evaluation

For the GPU, the operations are parallelised over individual elements to issue as many threads as there are elements in the vector. Each thread then simply computes the addition and writes the result back to the output vector in global memory. The performance of vector addition on the K20 GPU device is shown in Figure 6.2.

Size of vector	**Single prec.** (seconds)	**Speedup**	**Double prec.** (seconds)	**Speedup**
10,000	0.2	1.2X	0.3	1.0X
100,000	0.3	9.9X	0.9	4.1X
1,000,000	3.8	9.2X	7.2	4.9X

Table 6.2 *Vector addition on the K20 GPU, showing speedups over the fastest CPU times in each case.*

At the smallest size of 10,000 elements there is very little difference between CPU and GPU performance. However by 100,000 elements the GPU performance benefit is clear, especially at single precision floating-point which achieves a 10X speedup.

6.3 The dot product

The dot product is a good example of an operation that is trivial to perform on a CPU, but rather more difficult on a GPU device. Performing a sum reduction, that is summing together the result of each individual multiplication, is clearly an inherently serial procedure. Executing this on a highly parallel GPU requires an algorithm design that allows multiple threads to operate, as well as making use of shared memory to optimise data throughput [157].

6.3.1 Parallel algorithm design

Figure 6.1 shows the binary tree approach that is used to compute the dot product.

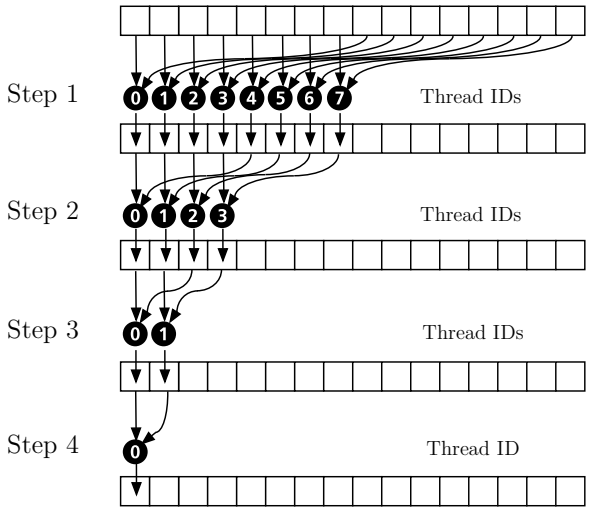


Figure 6.1 *Binary tree approach for sum reduction.*

Parallelization is possible by splitting the procedure into a series of partial summations. For vectors of length n, we first issue $n/2$ threads to calculate the summation of paired elements, resulting in $n/2$ summations. This is loaded into shared memory to avoid writing back to global memory. Then $n/4$ threads are issued to calculate the next level of partial sums, reading and writing to shared memory. This process continues until a final single thread computes the resulting sum, which can then be written to global memory. If the result is required on the CPU, then a further memory transfer is required.

6.3.2 Performance evaluation

This process is used in Nvidia's own linear algebra library CUBLAS, implemented with the `cublasDdot()` function [90]. This function is tested here on the Tesla K20 device, benchmarked against a simple single thread CPU implementation running on the i7 processor. Again this was compiled using -O3 optimisation at double precision, and the operation was performed in a loop over 44,100 steps. The result of the GPU function was not transferred to the host until the end of the iterations. The results are shown in Table 6.3.

Size of vector	Single thread i7 CPU (seconds)	CUBLAS K20 GPU (seconds)
10,000	0.4	1.0
50,000	1.8	1.2
100,000	3.7	1.5
1,000,000	20.2	3.3

Table 6.3 *Performance of the dot product on the i7 CPU and K20 GPU over 44,100 iterations, double precision floating-point.*

At the smallest 10,000 element size the CPU version is over twice as fast as the GPU. Between that level and 50,000 elements the GPU almost reaches a parity performance, and at 100,000 elements the GPU is now twice as fast. At one million elements the GPU achieves a 6X speedup.

6.4 Matrix by vector multiplication

The matrix by vector multiplication is the most common operation that will be needed in systems that implement embedded instruments. The standard matrix vector form of the finite difference scheme is written in terms of this operation, and it also features in the calculation of various types of linear system solutions (see Section 2.4.3).

The performance of matrix by vector multiplication is evaluated in two parts. Firstly, the relative efficiency of the CSR, DIA, and ELLPACK formats are assessed for single threaded CPU codes, and for GPU codes using Nvidia's CUSPARSE library function (which uses CSR) and custom written DIA and ELLPACK versions. Then, the DIA format version is used to compare single and multiple threaded CPU performance to that of the GPU version.

6.4.1 Performance comparison of matrix storage formats

Two matrices are used to compare the performance of the CSR, DIA, and ELLPACK formats. A five-band matrix consisting of a centre diagonal surrounded by two further bands above and below, and a thirteen-band matrix which has the following map:

```
int map[13] = {-15,-9,-8,-7,-2,-1,0,1,2,7,8,9,15}
```

These are tested at various sizes, from 1,000 rows up to 100,000 rows in the matrix and corresponding vector, again using a loop of 44,100 iterations.

CPU performance

Tables 6.4 and 6.5 shows the results at double precision floating-point running on the Intel i7 processor.

Row size	**CSR format** (seconds)	**DIA format** (seconds)	**ELLPACK format** (seconds)
1,000	0.2	0.3	0.2
10,000	1.7	2.9	2.4
50,000	10.4	14.4	12.3
100,000	21.2	28.7	24.6

Table 6.4 *Five band matrix by vector multiplication on the Intel i7 CPU.*

Row size	**CSR format** (seconds)	**DIA format** (seconds)	**ELLPACK format** (seconds)
1,000	0.6	0.7	0.6
10,000	4.8	6.7	5.8
50,000	29.9	34.2	32.9
100,000	60.3	68.6	65.2

Table 6.5 *Thirteen band matrix by vector multiplication on the Intel i7 CPU.*

Here the CSR format is outperforming both the DIA and ELLPACK formats across all matrix sizes. Despite the reduction in memory usage, the DIA format performs worse, around 1.5X slower than the CSR format. The ELLPACK format sits somewhere in-between the two. The main cause of the CSR performance is that despite using more data, it uses fewer floating-point operations to calculate each multiplication. The DIA and ELLPACK formats require more operations as their data is decomposed, and DIA requires the column to be calculated from the integer map.

GPU performance

The relative GPU performance of the storage formats is assessed using Nvidia's own CUSPARSE library to perform the CSR format computation, and custom written DIA and ELLPACK versions. Again the calculations are performed for 44,100 iterations, and do not include copying results back from GPU. The DIA and ELLPACK formats use column major data table decomposition to facilitate memory coalescing. The operations are parallelised over the rows of the matrix, so each thread computes the final summation value for a given row. A column-major decomposition of the data table for the DIA and ELLPACK formats ensures that consecutive threads are reading consecutive data elements from global memory. Tables 6.6 and 6.7 show the results.

Row size	**CSR CUSPARSE** (seconds)	**DIA format** (seconds)	**ELLPACK format** (seconds)
1,000	0.6	0.3	0.4
10,000	0.7	0.4	0.5
50,000	2.7	1.3	1.6
100,000	4.9	2.3	2.9

Table 6.6 *Five band matrix by vector multiplication on the K20 GPU.*

Row size	**CSR CUSPARSE** (seconds)	**DIA format** (seconds)	**ELLPACK format** (seconds)
1,000	0.7	0.6	0.6
10,000	1.5	0.7	1.0
50,000	6.1	2.7	3.1
100,000	11.7	4.9	5.9

Table 6.7 *Thirteen band matrix by vector multiplication on the K20 GPU.*

Both sets of tests were computed at double precision floating-point. Compared to the previous CPU results, the situation is now reversed. The DIA format now outperforms the CSR format by 2X, with the ELLPACK format also considerably more efficient than the CSR. This clearly demonstrates the value of reducing data usage for GPU operations, where memory bandwidth is the critical factor.

6.4.2 Single and multi-threaded CPU performance of DIA format

Having established that the DIA format has considerable efficiency benefits for GPU operation, this section gives a more thorough examination of the CPU to GPU performance. Single and multi-threaded CPU code is tested to produce comparable benchmarks for CPU performance, and these are used to assess the speedups achievable with the GPU version.

The CPU version was tested on both the Intel i7 and Xeon processors, using various numbers of threads. Both the five-band and thirteen-band matrices were used, across the same range of sizes as before. Table 6.8 shows the results for five-band matrix, and Table 6.9 the thirteen-band.

Process	**size: 1,000** (seconds)	**size: 10,000** (seconds)	**size: 50,000** (seconds)	**size: 100,000** (seconds)
i7 - 1 thread	**0.3**	**3**	14	28
i7 - 4 threads	2	4	**7**	**12**
i7 - 8 threads	4	5	8	**12**
Xeon - 1 thread	0.5	5	23	46
Xeon - 6 threads	6	8	17	19
Xeon - 12 threads	7	10	20	21

Table 6.8 *Five band matrix by vector multiplication, DIA format.*

Process	**size: 1,000** (seconds)	**size: 10,000** (seconds)	**size: 50,000** (seconds)	**size: 100,000** (seconds)
i7 - 1 thread	**0.7**	7	34	69
i7 - 4 threads	3	**5**	**14**	31
i7 - 8 threads	5	6	15	**29**
Xeon - 1 thread	1	11	56	122
Xeon - 6 threads	8	11	27	53
Xeon - 12 threads	15	17	27	54

Table 6.9 *Thirteen band matrix by vector multiplication, in DIA format.*

Again, these are computed at double precision floating-point, and in a loop over 44,100 iterations. For up to 10,000 rows the single thread version is most efficient in each case, with the i7 performing best with its greater clock frequency. However at 50,000 rows and above the multi-thread versions achieve considerably better efficiency.

It should be noted that the timings for the Xeon processors may be improved for the larger sizes by using an OpenMP threading architecture. As the Xeon hardware consists of a dual-socket with two processors, the parallel allocation of memory can produce further efficiency gains when using a larger number of threads by allocating the individual parts of the memory directly to the system for the core. However, for the purposes of the modelling tested in Chapter 7, the matrix row sizes are in the order of twenty to thirty thousand rows. At this relatively small size the application of a large number of threads across both CPUs is unlikely to produce further speedups.

6.4.3 GPU performance comparison of DIA format

The CPU results can now be compared to the GPU performance across the same tests, using the fastest CPU times as the benchmark in each case. The Tesla K20 device is used, at single and double precision. The results for both the five-band and thirteen-band matrices are shown in tables 6.10 and 6.11.

Row size	**Single prec.** (seconds)	**Speedup**	**Double prec.** (seconds)	**Speedup**
1,000	0.2	1.4X	0.3	1.0X
10,000	0.4	7.3X	0.5	5.0X
50,000	1.0	7.0X	1.3	5.2X
100,000	1.6	7.5X	2.3	5.3X

Table 6.10 *Five band matrix by vector multiplication on the K20 GPU, showing speedup over fastest CPU times.*

At the small 1,000 row size there is little to choose between the fastest CPU performance (single thread) and the GPU performance. However at 10,000 rows and above the GPU outperforms the fastest CPU results by margins of 5X to 6X at double precision, at 7X to 8X at single precision.

Row size	Single prec. (seconds)	Speedup	Double prec. (seconds)	Speedup
1,000	0.5	1.3X	0.6	1.0X
10,000	0.7	7.1X	0.7	5.0X
50,000	1.9	7.3X	2.7	5.1X
100,000	3.4	8.4X	4.9	5.9X

Table 6.11 *Thirteen band matrix by vector multiplication on the K20 GPU, showing speedup over fastest CPU times.*

6.5 Matrix vs non-matrix forms

The previous sections have examined the performance of sparse matrix operations that occur when designing complex simulations with embedded systems. However, the matrix vector formation is not always necessary, depending on the category of system that is used.

6.5.1 Categories of schemes

Recall the generalised form of the finite difference schemes from Section 2.4.

$$\mathbf{A}\mathbf{u}^{n+1} = \mathbf{B}\mathbf{u}^{n} + \mathbf{C}\mathbf{u}^{n-1} \tag{6.2}$$

where $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ are sparse update matrices. The resulting systems can be grouped into four categories:

1. Uniform explicit schemes.
2. Non-uniform, but explicit schemes.
3. Implicit schemes, with constant update matrices.
4. Implicit schemes, with update matrices constructed at each time step.

In the first category are explicit schemes where the update matrices $\mathbf{B}$ and $\mathbf{C}$ contain bands of constant coefficients. They are often of Toeplitz or block Toeplitz form. For such systems, the matrix form can be unrolled so that grid points can be updated by a simple equation using scalar coefficients and neighbouring points, as in the three-dimensional case from Chapter 3. Whilst the matrix vector form is useful for designing and prototyping complex systems, it is important to understand the performance

benefits of the 'unrolled' formation compared to the matrix form. A simple 2D wave equation system is examined for this purpose.

6.5.2 2D wave equation system

The 2D wave equation scheme is a linear system which can be expressed as an explicit update equation as

$$u_{l,m}^{n+1} = (2 - 4\lambda^2)u_{l,m}^{n} + \lambda^2(u_{l+1,m}^{n} + u_{l-1,m}^{n} + u_{l,m+1}^{n} + u_{l,m-1}^{n}) - u_{l,m}^{n-1} \tag{6.3}$$

Fixed boundary conditions are used for this test case. The scheme can be written in matrix vector form as

$$\mathbf{u}^{n+1} = \mathbf{B}\mathbf{u}^{n} - \mathbf{u}^{n-1} \tag{6.4}$$

where the coefficients $(2 - 4\lambda^2)$ and λ^2 form five bands in the block Toeplitz update matrix $\mathbf{B}$, as shown in Figure 6.2 .

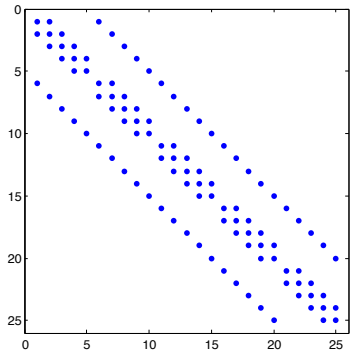


Figure 6.2 *Sparsity pattern of block Toeplitz matrix* $\mathbf{B}$.

The wave speed c is calculated from the physical parameters of tension and surface density for the simulation, with the grid spacing then set at the Courant limit for the scheme. A sample rate of 44.1kHz is used, and the simulation is computed for 44,100 time steps. Varying the physical size (in metres) of the system varies the number of grid points used in each case.

6.5.3 CPU performance evalutaion

The single thread CPU performance running on the Intel i7 processor is shown in Table 6.12, for the unrolled non-matrix case as well as DIA and CSR matrix implementations.

Physical size	**Non-matrix** (seconds)	**Speedup** (over CSR)	**CSR format** (seconds)	**DIA format** (seconds)
1.3m × 1.7m	5.1	3.3X	16.7	28.7
2.3m × 2.7m	15.2	3.6X	55.3	83.9
4.3m × 4.7m	51.1	3.9X	201.1	290.1
8.3m × 8.7m	192.2	3.7X	714.6	1024.2

Table 6.12 *2D wave equation solver at 44.1kHz, single thread i7 CPU.*

These are all computed at double precision floating-point. As before, the CSR format performs better than the DIA format when using the CPU due to the reduced operation count. However, the unrolled version is up to 3.9X faster than the CSR matrix version.

6.5.4 GPU performance evalutaion

For the GPU comparison the Nvidia CUSPARSE library function is used to compute the matrix by vector multiplication, along with the custom DIA format function from previous sections. Table 6.13 shows the results at double precision for the unrolled and matrix versions.

Physical size	**Non-matrix** (seconds)	**Speedup** (over DIA)	**DIA format** (seconds)	**CSR CUSPARSE** (seconds)
1.3m × 1.7m	0.9	2.4X	2.2	4.3
2.3m × 2.7m	1.9	2.8X	5.4	11.1
4.3m × 4.7m	5.2	3.2X	16.4	34.9
8.3m × 8.7m	17.9	3.2X	57.9	123.6

Table 6.13 *2D wave equation solver at 44.1kHz on the K20 GPU.*

This again demonstrates the benefits of the DIA format for GPU operation, giving a 2X speedup over the CUSPARSE CSR version. The unrolled version is still optimal, showing a 3X speedup over the fastest matrix form.

6.6 Summary

This chapter has compared the performance of basic linear algebra operations on both CPU and GPU processors, examining the use of various sparse matrix storage formats. The results from this chapter can be summarised as follows:

1. *Vector addition*: CPU code only benefits from multi-threading when the size of the vectors is above 100,000 elements. The GPU code gives speedups over the CPU version at 10,000 elements and above, around 5X at double precision, and 10X at single precision.

2. *Dot product*: GPU code is more efficient for vectors over 10,000 elements in size, reaching a 6X speedup over the CPU version for vectors containing one million elements.

3. *Sparse matrix formats*: Considering the matrix by vector multiplication, the CSR format is more efficient than DIA or ELLPACK on the CPU, performing at 1.5X over DIA. However, on the GPU the DIA format is most efficient, showing 2X speedups over the CSR format used in the CUSPARSE library. Note that these timings are specific to the matrix formats.

4. *Matrix by vector multiplication*: Multi-threaded code is more efficient for matrix by vector multiplication using the DIA format at a row size of 10,000 and above. The GPU version shows 5X speedup over the CPU version at double precision, 7X at single precision.

5. *Matrix vs non-matrix forms*: The unrolled non-matrix update form is 3.7X faster than matrix form on the CPU, and 3X on the GPU, for performing a 2D wave equation system.

The following chapter examines the usage of these operations in an integrated simulation for three-dimensional physical modelling synthesis.

Chapter 7

An integrated model of the timpani drum

The timpani drum requires a full three-dimensional model that can serve as a suitable test case for examining the parallel computation techniques outlined in the previous chapters of this thesis. The objective is to create a complete model of the instrument for the purpose of sound synthesis, including the user control, and then compute the simulation in as short a time as possible. By embedding the drumhead and shell of the timpani inside a three-dimensional space, the complete acoustic field of the instrument can be computed over spatial grids which are updated over time. The vibrations of the membrane are coupled with the surrounding air space above and below, as shown in Figure 7.1. Obviously one would like to be able to interact with the model in *real time*, but this is still unrealistic using currently available hardware. However, GPUs can achieve useful acceleration over CPU computation, and so can improve the usability of such systems.

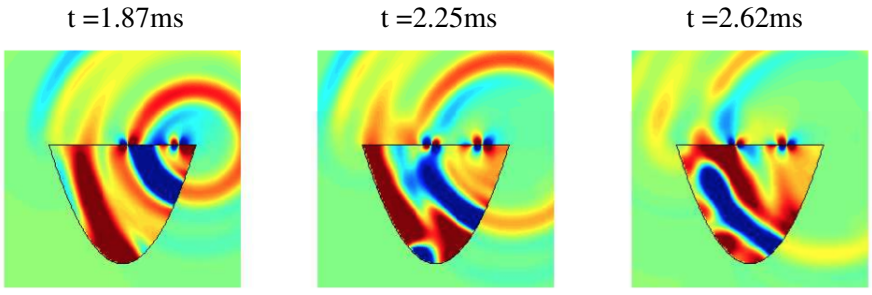


Figure 7.1 *Snapshots of the acoustic field progressing over time for a 3D model of a timpani drum embedded in an anechoic space.*

This chapter begins with an overview of the timpani drum model, followed by a detailed description of the computational elements of the system. The implementation of the model is then considered using various sparse matrix storage formats, as well as a 'matrix-free' version that unrolls some of the calculations in order to reduce the memory bandwidth requirements. A test case simulation is used to compare CPU performance to that of the GPU, with a specific analysis of the speedups obtained by each element of the system. Finally, a large-scale environment is described that allows multiple timpani drums to be embedded in a simulation of a hall or concert space.

7.1 Overview of the model

The timpani drum has two major components; a drumhead and a rigid shell. The drumhead is an elastic membrane, the tension of which determines the fundamental pitch of the instrument and can be varied within a given range by a foot pedal. This membrane is stretched over a rigid shell enclosure, which has a bowl profile. Timpani drums typically come in a range of four sizes, from 23 to 32 inches in diameter [158]. The model of the timpani drum described here embeds these two elements inside a small three-dimensional air space (Figure 7.2). An absorbing condition is used at the external boundaries of this space to create an anechoic setting for playing the instrument. Larger acoustic environments are considered in Section 7.5, but a small 1m^3 anechoic space is the most efficient setting from a computational perspective.

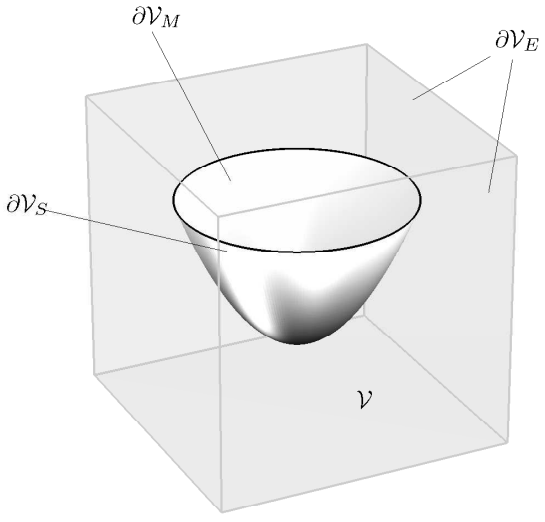


Figure 7.2 *Timpani drum geometry, indicating the computational space $\mathcal{V}$, its external boundary $\partial\mathcal{V}_E$, membrane boundary $\partial\mathcal{V}_M$ and boundary with the rigid shell $\partial\mathcal{V}_S$.*

The drumhead is modelled using a nonlinear membrane that allows pitch glide phenomena under high striking amplitudes. This uses a simplification of the von Kármán system as detailed by Berger [159]. This model is similar to that used by Rhaouti et al. [160], and for a modal implementation of coupled membranes by Avanzini [161]. It has also been demonstrated for simulations of the snare drum [162].

The acoustic field is modelled using the 3D wave equation. For testing purposes the lossless form is used, although for larger environments the viscosity version is applied (Section 5.3). A simple resistive boundary is employed as described in Section 5.1.1, with a high absorption coefficient giving a usable anechoic environment. The rigid shell of the drum is assumed to be parabolic for ease of calculation, and is modelled using a Neumann (zero normal velocity) condition on the acoustic field either side of its boundary, given by

$$\mathbf{n}_S \cdot \nabla_{3D} \Psi = 0 \qquad \text{over} \qquad \partial \mathcal{V}_S \tag{7.1}$$

where $\mathbf{n}_S$ is the normal to the shell surface.

The membrane is defined over a two-dimensional region $\partial \mathcal{V}_M$, and its dynamics are described by the following equation

$$\begin{aligned} \frac{\partial^2 w}{\partial t^2} &= c_M^2 \left(1+g\right) \nabla_{2D}^2 w - \kappa^2 \nabla_{2D}^2 \nabla_{2D}^2 w + c_M^2 \alpha \nabla_{2D}^2 \frac{\partial w}{\partial t} \\ &\quad + \frac{1}{\rho_M} \left(f_+ + f_-\right) + \frac{1}{\rho_M} \delta(x - x_0, y - y_0) f_{exc} \end{aligned} \tag{7.2}$$

where $w(x, y, t)$ is the transverse displacement over time t and coordinates x and y, and ρ_M is the surface density of the membrane. The wave speed c_M and stiffness parameter κ are defined as

$$c_M = \sqrt{T_0 / \rho_M} \qquad \kappa = \sqrt{EH^3 / 12 \rho_M (1 - \nu^2)} \tag{7.3}$$

where H is the membrane tickness, in m, E is Young's modulus, in kg /s^{2}m, T_0 is membrane tension/unit length, in kg/s^2, and ν is Poisson's ratio (dimensionless). α is a parameter that determines viscothermal damping effects. The 2D Laplacian operator is defined as

$$\nabla_{2D}^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \tag{7.4}$$

The factor g represents an averaged nonlinearity, and is given by

$$g = \frac{EH}{2T_0(1-\nu^2)} \int \int_{\partial \mathcal{V}_M} |\nabla_{2D} w|^2 d\sigma \tag{7.5}$$

The term $f_{exc}(t)$ is a forcing function acting at position (x_0, y_0) on the membrane, and δ is a 2D Dirac delta function. A raised cosine excitation is used for which amplitude and duration may be adjusted, corresponding to the strike of a mallet.

The system contains terms that couple the membrane to the acoustic space, both above and below. If the velocity potentials on the upper and lower faces of the membrane are defined as Ψ_+ and Ψ_-, the conditions may be written as

$$f_- = \rho \frac{\partial \Psi_-}{\partial t} \qquad f_+ = -\rho \frac{\partial \Psi_+}{\partial t} \tag{7.6}$$

and

$$\frac{\partial w}{\partial t} = -\mathbf{n}_{M,+} \cdot \nabla \Psi_+ = \mathbf{n}_{M,-} \cdot \nabla \Psi_- \tag{7.7}$$

where $\mathbf{n}_{M,+} = -\mathbf{n}_{M,-}$ are the 3D outward unit normals to the membrane in the positive and negative directions.

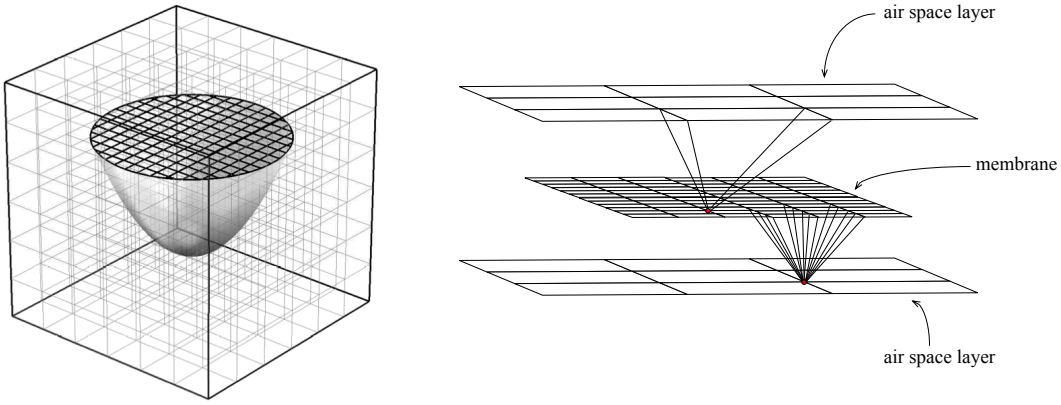


Figure 7.3 *Computational grids used in the timpani model. The more dense membrane grid sits in-between two layers of the 3D air space grid. Interpolation from the air space to the membrane uses the four surrounding points, but the reverse interpolation requires many surrounding points from the membrane.*

The finite difference schemes used to approximate this model make use of regular grids. The air space uses the basic 3D scheme as detailed in Section 2.1.4. The membrane grid effectively sits in-between two layers of the computational grid for the air space and the circular shape is achieved using a simple staircase approximation at the boundary over the regular grid. For typical instrument designs, note that the discretisation leads to state grids as shown in Figure 7.3, where the membrane has a more dense grid than the air space. See [163] for a full description of the numerical schemes used for this simulation.

7.2 Computational elements

The simulation model is broadly divided into two parts. First is a series of operations that set up the necessary components of the system, such as creating and initialising memory for the state grids, and creating various matrices and other necessary variables. Then, a further series of operations occur inside the time iteration loop that update the state grids for each given time step of the simulation. All of the simulations are computed here at full audio rates. The setup work here is very small when compared to the main time loop (typically much less than a second for a single thread on a CPU), and so is not subjected to parallelisation.

7.2.1 Simulation setup

The simulation setup begins with a definition of the physical parameters of the system, such as the radius and tension of the membrane, and the details of the strike amplitude and position, as detailed in Algorithm 16. The main calculation required is to build a list that enumerates the boundary points in the three-dimensional space that will define the rigid shell of the drum (Figure 7.4).

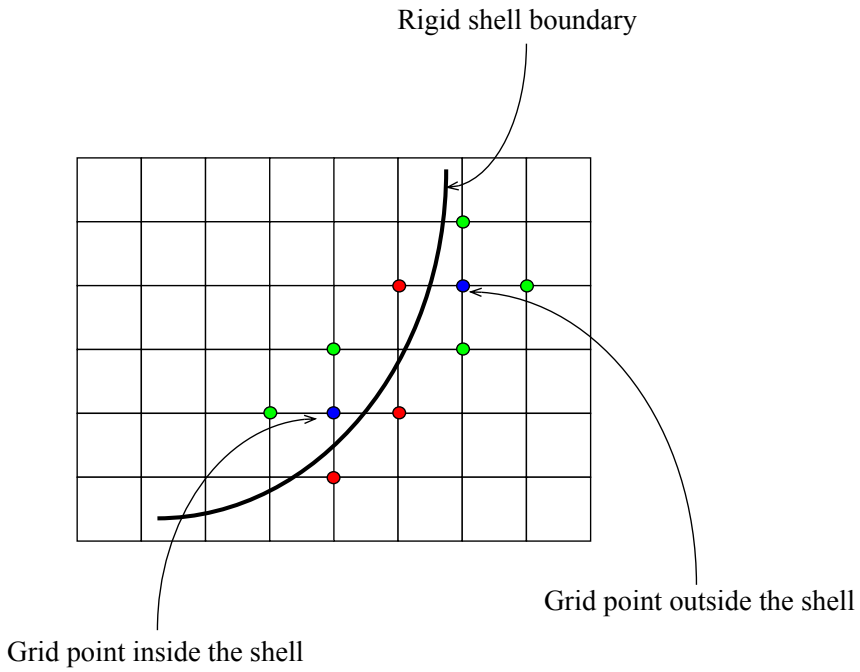


Figure 7.4 *2D cross-sectional view of the timpani shell in the simulation space. Blue indicates a centre point of a stencil, green are neighbour points that are inside a boundary, and red those that fall outside.*

This list contains the details required to build the Laplacian operator at the boundaries of the shell. Specifically it stores details of which of the stencil legs of the scheme fall inside or outside of the calculated parabolic shape for grid points at the boundary layer edge. This list can contain many thousands of entries. Having allocated and initialised memory for the three-dimensional air space states, the membrane setup consists of defining a series of initial matrices alongside state vectors that will be used in the time iteration.

Algorithm 16 *Simulation Setup*

Set global input parameters:
SR ← sample rate
Tf ← duration (s)
c ← wave speed of air (m/s)
ρ_0 ← air density (kg/m^3)
Set depth of timpani bowl and size of simulation space
Set read output locations
Set membrane parameters:
ρ ← density (kg/m^3)
T_0 ← tension (N/m)
R ← radius (m)
E ← Young's modulus (Pa)
H ← thickness (m)
ν ← Poisson's ratio
α ← loss parameter
Set strike parameters:
dur ← duration
$famp$ ← amplitude
Build list of rigid shell boundary points
Create and initialise memory
Create membrane and interpolation matrices

The remaining setup operations create the interpolation matrices, which are described in more detail in the following section.

7.2.2 Time iteration matrices

Whilst the simulation of the three-dimensional acoustic space can be computed as a simple explicit update equation applied to each grid point (Chapter 3), the calculation of the membrane (when coupled to the acoustic field) requires a series of more complex operations. These operations are most easily described in a matrix form (Section 2.4), with the two-dimensional membrane state decomposed into vectors which are acted upon by these matrices. The time iteration therefore requires a set of persistent matrices that can be computed before the time loop begins.

Membrane matrices

Figures 7.5 and 7.6 show the main system matrices required for calculating the drumhead membrane of the timpani. The sizes shown are for a test case 29inch timpani drum at a sample rate of 44.1kHz, as detailed in Section 7.4.1.

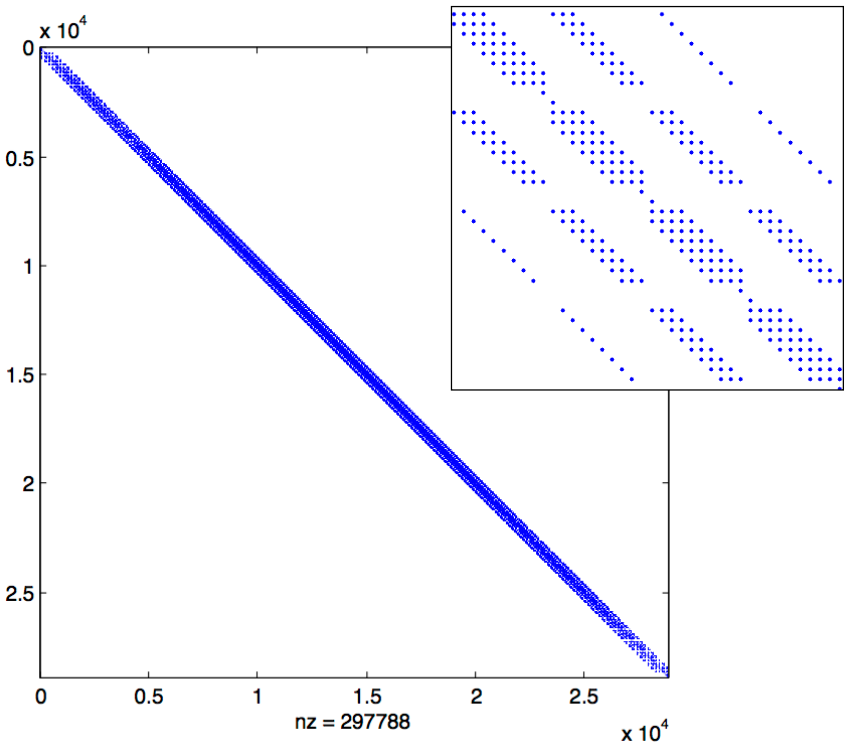


Figure 7.5 *Membrane state matrix B, with magnified view of the sparsity pattern.*

These matrices are the basis of the linear membrane update equation (over displacement, $\mathbf{u}$) in matrix form (Section 2.4), given in isolation by

$$\mathbf{u}^{n+1} = \mathbf{B}\mathbf{u}^{n} + \mathbf{C}\mathbf{u}^{n-1} \tag{7.8}$$

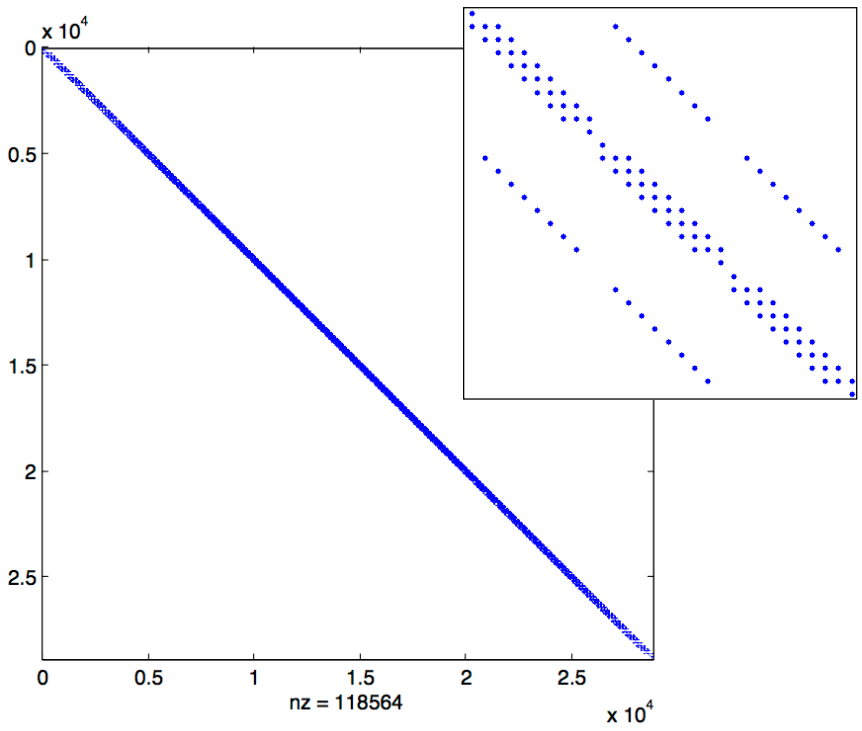


Figure 7.6 *Membrane state matrix C, with magnified view of the sparsity pattern.*

The state matrix $\mathbf{B}$ contains thirteen diagonal bands (due to the biharmonic operator), whilst the state matrix $\mathbf{C}$ contains five diagonal bands (the standard Laplacian). These matrices are largely block Toeplitz, but are also affected by a circular masking operation. The circular shape of the membrane is achieved using a staircase approximation at the boundaries, and this is implemented using a circular mask placed over a square grid. This can be seen in the more sparsely populated areas at either end of the diagonals bands. A further state matrix based on a scaled Laplacian operator is also required, which will be referred to as matrix $\mathbf{V}$.

Interpolation matrices

The energy conserving coupling between the membrane grid and the coarser layers of the air space grid above and below are most easily described with the use of two interpolation matrices. The interpolation from the 3D air layer to the 2D membrane is denoted here by the matrix $\mathbf{I}^{32}$, whilst the reverse direction interpolant is the transpose of this matrix, denoted by $\mathbf{I}^{23}$, for reasons relating to numerical stability via energy conservation. The matrices are shown in Figure 7.7. These are rectangular in shape due to the difference in the densities of the grids for the membrane and the air space

layers.

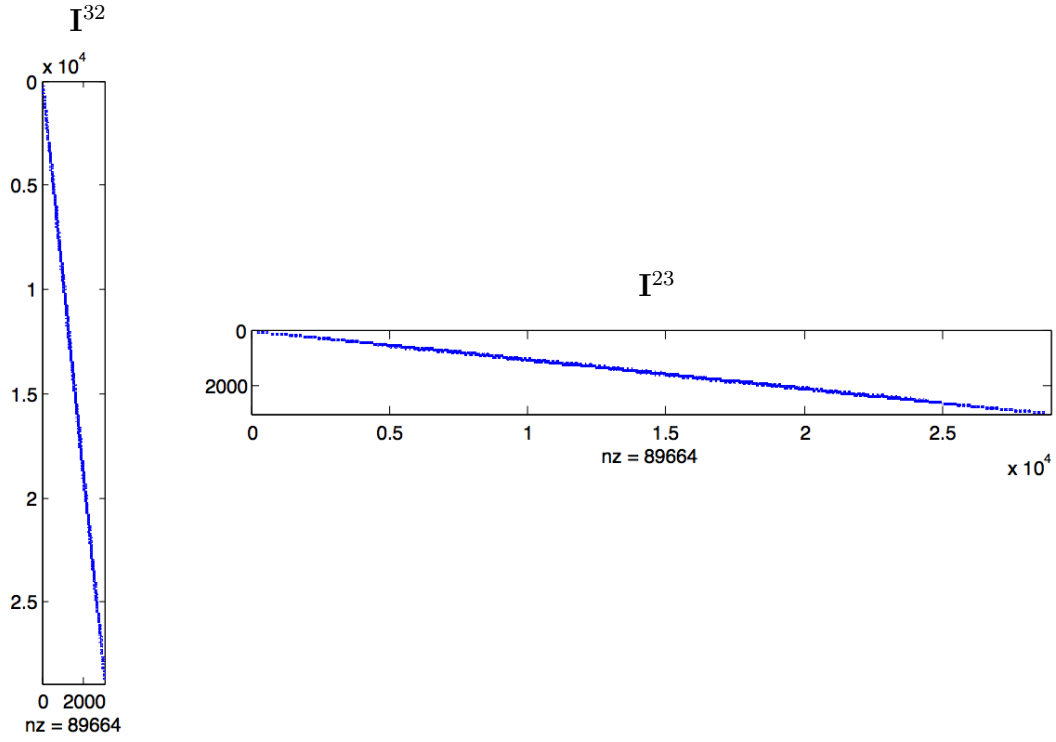


Figure 7.7 *Interpolation matrices,* $\mathbf{I}^{32}$ *and* $\mathbf{I}^{23}$.

The matrix $\mathbf{I}^{32}$ contains four elements on each row (for grid points in the interior), containing the standard weightings for a bilinear interpolation. The transposed matrix, however, contains many more elements on each row. The application of these matrices in the time iteration loop often appears as the product of $\mathbf{I}^{32}$ with $\mathbf{I}^{23}$. This creates a matrix as shown in Figure 7.8.

However, this multiplication of $\mathbf{I}^{32}$ with $\mathbf{I}^{23}$ has an adverse effect in the sparsity pattern of the resulting matrix. Performing a matrix by vector multiplication using this combined matrix is clearly much less efficient than performing two separate matrix by vector multiplications using the individual factored matrices. Even with the benefit of a parallel implementation on the GPU, this would still be far less efficient. Therefore, all of the applications of the interpolants in the test simulation make use of the individual matrices.

A further benefit of using the individual versions is that it is possible to 'unroll' their application in a matrix by vector multiplication to create a matrix-free operation. This has implications for GPU memory bandwidth, and is detailed further in Section 7.3.5.

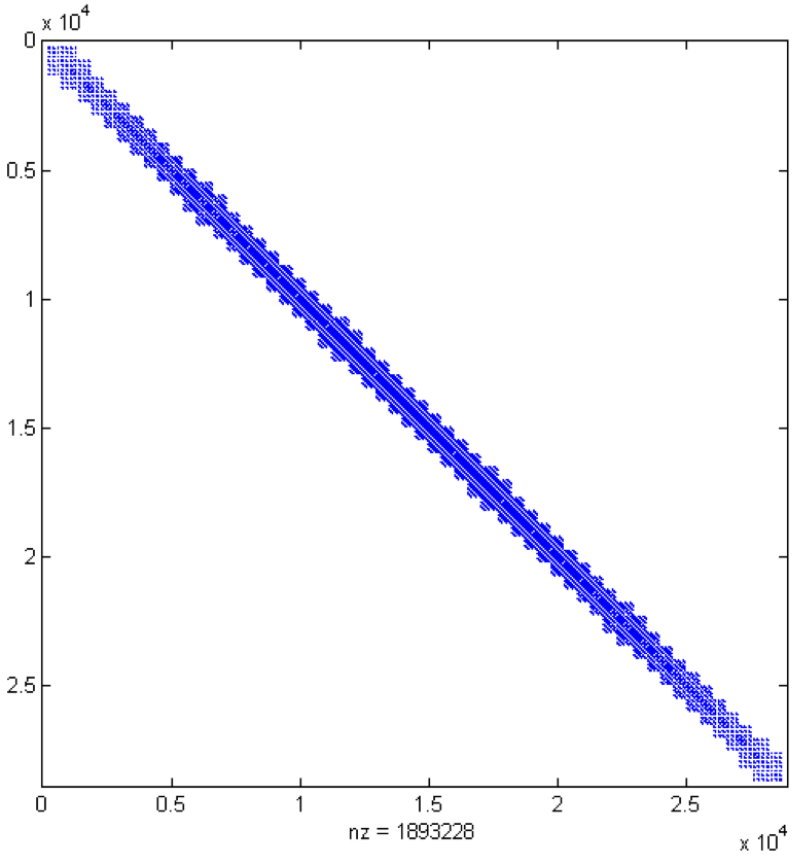


Figure 7.8 *Combined interpolation matrix* $\mathbf{I}^{32} \cdot \mathbf{I}^{23}$.

7.2.3 Time iteration operations

The operations performed at each iteration of the time loop can be grouped into seven stages:

1. Calculate the 3D Laplacian, in the absence of any shell boundaries.
2. Adjust the 3D Laplacian to include the rigid shell boundaries.
3. Calculate the update for the drumhead membrane, and insert the input.
4. Perform a linear system solution using the membrane vectors and the interpolation matrices.
5. Update the three-dimensional air space, using the previously calculated Laplacian.
6. Adjust the layers above and below the membrane to incorporate its effect.
7. Read the output, and swap memory pointers.

Each of these is examined in detail in the following sections.

Stage 1. Calculate the 3D Laplacian

The update for the three-dimensional air propagation is split into two parts. First the Laplacian operator is calculated here, and then at Stage 5 the remaining calculations are performed. This stage uses three loops, over the X, Y and Z dimensions of the air space, and for each point it computes the Laplacian and stores the result in one of the three state arrays used for the air propagation. The process also leaves a halo layer around the edge of the 3D data, in order to implement the first order absorbing boundary condition.

Stage 2. Adjust the 3D Laplacian

In this stage the previously calculated Laplacian data is adjusted to incorporate the rigid shell boundary of the timpani in the three-dimensional space. The grid points and Laplacian data required to implement the shell are stored in a list that is computed at the setup phase (see Section 7.2.1). Here, we iterate through that list and adjust the state data to include this new Laplacian calculation. This stage requires less computation than Stage 1, but still has a material effect on the overall time.

Stage 3. Calculate the update for the drumhead membrane

Calculating the update for the membrane is performed using a five step process, as detailed in Algorithm 17 (lower case bold characters denote vectors, and upper case denote matrices).

Algorithm 17 *Calculate the drumhead membrane*

1: $\mathbf{a} \leftarrow \mathbf{V} \cdot \mathbf{u}^n$
2: $a_{inv} \leftarrow 1/(1 + \mathbf{a}^T \cdot \mathbf{a})$ ▷ a_{inv} is a scalar
3: $\mathbf{g} \leftarrow \mathbf{B} \cdot \mathbf{u}^n + (\mathbf{C} + \mathbf{I}^{32} \cdot \mathbf{I}^{23}) \cdot \mathbf{u}^{n-1} - \mathbf{a} \cdot (\mathbf{a}^T \cdot \mathbf{u}^{n-1})$
4: $\mathbf{b} \leftarrow$ Calculated using data from the layers of air space above and below the membrane, and the stored Laplacian.
5: $\mathbf{r} \leftarrow \mathbf{g} - \mathbf{I}^{32} \cdot \mathbf{b}$

The first steps are a simple matrix by vector multiplication, followed by a dot product to compute a scalar value that is used by the linear system solution later on (this corresponds to the tension modulation effect). Step three contains the most calculations, with four matrix by vector multiplications, a dot product, and vector scaling and

addition. At step four a new vector is computed based on the surrounding air space data, which involves iterating over a small 2D area and performing some addition and scaling operations. Finally, at step five there is another matrix by vector multiplication, and vector subtraction. Overall, Stage 3 contains a substantial amount of computation, and represents a significant percentage of the total time for a single time step of the simulation.

Stage 4. Linear system solution

A linear system solution is required to incorporate the interpolation matrices into the update of the drumhead membrane. Despite belonging to the fourth category of schemes detailed in Section 6.5.1, the linear system can be separated into a constant sparse linear system solution corresponding to the interpolation, and then a very simple linear solution involving a rank one perturbation of the identity. This has an efficient implementation via methods such as the Woodbury inversion formula [116], and is a consequence of the very simple form of the nonlinearity.

The system matrix for this solver is the combination of the two interpolants, $\mathbf{I}^{32} \cdot \mathbf{I}^{23}$, and can be solved using a modified Jacobi-type iteration. Note, however, that the factored individual matrices will be used in practice. The iterative method is shown in Algorithm 18. This consists of an initial matrix by vector multiplication using the two interpolants, and a resulting vector subtraction. Then a dot product operation, vector scaling, and a further vector subtraction are performed.

Algorithm 18 *Linear system solution for the timpani simulation*

1: **for** $i = 1 : iterations$ **do**
2: $\quad \mathbf{p} \leftarrow \mathbf{r} - \mathbf{I}^{32} \cdot (\mathbf{I}^{23} \cdot \mathbf{u}^{n+1})$
3: $\quad \mathbf{u}^{n+1} \leftarrow \mathbf{p} - \mathbf{a} \cdot (\mathbf{a}^T \cdot \mathbf{p}) \cdot a_{inv}$
4: **end for**

This method converges to machine accuracy in a small number of iterations. Whilst the interpolation matrices contain fewer entries than the membrane matrices used at Stage 3, the repeated application of these in a loop results in this being the most computationally expensive stage of the entire simulation.

However, referring to the results obtained in Chapter 6, we expect to be able to accelerate these operations on the GPU. Matrix by vector multiplication, and the dot product and subtraction, are all amenable to parallel computation. One caveat is that

the interpolation matrices are not diagonally banded, and so the DIA format will not be useful in this case. See Section 7.3 for further detail.

Stage 5. Update the air space

This stage completes the three-dimensional air space update, using the previously computed Laplacian data. It consists of three loops over the X, Y, and Z dimensions, as at Stage 1. At the boundary edge layer, loss coefficients are applied to compute the absorbing boundaries. See Section 7.1 for details of the scheme and boundary condition.

Stage 6. Adjust the 3D layers

The final update to the air space is to adjust the two layers of data above and below the membrane, to incorporate the membrane interaction with these layers. First a matrix by vector multiplication is performed using the membrane data and the $\mathbf{I}^{23}$ interpolation matrix. Then a loop over the small area of the layers performs a final calculation on each data element.

Stage 7. Read output

One of the key advantages of modelling the entire three-dimensional acoustic field over time is the degree of flexibility this gives in terms of reading the output of the simulation. At the most basic level, data from an individual grid point in the acoustic field can be saved into an output vector, and when normalised can be played back as a standard digital audio signal. Data from two different grid points can be saved into output vectors to provide a stereo track, or indeed more points can be used to produce a multi-channel rendering. A simple stereo output is used in this test case scenario.

7.3 Implementation designs

The operations performed at each of the seven stages involve either explicit updates where (for a serial algorithm) loops are used to apply a calculation to a series of data elements, or sparse matrix by vector operations, or sometimes a combination of the two. Therefore, an implementation in C or CUDA code will require the use of sparse matrix storage formats and related functionality.

Multiple implementations of the timpani drum simulation are examined here to compare the efficiency of the various matrix storage formats detailed in Chapter 6 as

applied to a complex model. Implementations using the CSC/CSR formats, as well as customised DIA and ELLPACK, are considered. The particular form of the matrices being used also allows for 'unrolling' of some of the operations to give a matrix-free version, as detailed in Section 7.3.5.

7.3.1 Parallel implementation of the time iteration stages

In seeking to accelerate a complex simulation using GPU programming, it is first necessary to understand the suitability of the various stages of the simulation to a highly parallel architecture. Issues such as minimising transfers between host and device, and achieving coalesced memory transfers from global GPU memory also need to be considered at the initial design phase.

Stage 1. Calculate the 3D Laplacian

This initial calculation is a straightforward update over the three-dimensional data, and so can be easily implemented using the techniques outlined in Chapter 3. At a sample rate of 44.1kHz, the state data for a cubic metre space is held in vectors of size 420 thousand elements. Whilst this is only a relatively small data set when compared to larger room acoustics models, it is still large enough to expect good acceleration on the GPU.

One potential issue is mapping thread blocks to the three-dimensional data, which for example would measure $75 \times 75 \times 75$ grid points at a sample rate of 44.1kHz. The typical size of the thread block in CUDA is 32×8 or $32 \times 4 \times 2$, as used in the test case simulations in Chapter 3. The data set is therefore not an integer multiple of the thread block.

The basic approach for implementing this in CUDA is to issue an over-sized number of thread blocks in the thread grid, and then use a conditional statement to ensure that only the required data is updated. A further optimisation is to make use of the *cudaMallocPitch* functionality to allocate padded memory space that guarantees coalesced transfers. Tests on the small data set used here showed no efficiency benefit using this padded memory space, possibly due the requirement of using a conditional statement to implement the halo layer for the boundary condition.

Stage 2. Adjust the 3D Laplacian

Adjusting the Laplacian for the rigid shell boundary involves reading through a table of integer data entries. Each row entry details a grid point in the three-dimensional air space and the Laplacian information that will implement the boundary condition at that point. An example of the data set is shown in Table 7.1.

X	Y	Z	East	West	North	South	Up	Down	Sum
12	38	40	0	1	1	1	1	1	-5
12	38	41	0	1	1	1	1	1	-5
12	38	42	0	1	1	1	0	1	-4
12	38	43	1	0	1	1	1	0	-4
12	38	44	1	0	1	1	1	1	-5
12	38	45	1	0	1	1	0	1	-4
12	38	46	1	1	1	1	1	0	-5
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 7.1 *Rigid shell boundary list. A column-major decomposition in linear memory allows for coalesced memory transfers on the GPU.*

Each row can be processed independently, and so the table can be computed by simply issuing the appropriate number of threads on the GPU. Being a table, there are clearly two orientations that can be used to decompose the data into linear memory for use in CUDA, either row or column major. By choosing a column-major approach, each thread on the GPU will be reading consecutive integer values from memory, and so coalesced transfers should occur. A row-major approach would not allow this, and so would be far less efficient.

Stage 3. Calculate the update to the drumhead membrane

Referring to Algorithm 17, each of the five steps required to calculate the update to the drumhead membrane consists of operations that can be parallelised. The sparse matrix by vector calculations, vector additions, and dot products have all been examined in Chapter 6. Note that results of the sum reductions need to be kept on the GPU, not transferred back to the host, as they are required as scalars for further vector operations.

Stage 4. Linear system solution

This stage consists entirely of matrix by vector multiplications, additions, and a dot product, and so represents no difficulties other than the non-diagonal structure of the interpolation matrices, and the small row size of the $\mathbf{I}^{23}$ matrix.

Stage 5. Update the air space

This update is again a loop over the three-dimensional data set, and so can be parallelised as per Stage 1.

Stage 6. Adjust the 3D layers

The initial matrix by vector multiplication is straightforward, but the iteration over the small data area of the layers above and below the membrane does not allow a large number of threads to be issued. This limits the level of acceleration possible at this stage.

Stage 7. Read output

Reading the dual vector stereo output is achieved with a single thread on the GPU. Note that the output vectors are stored on the GPU during the simulation, and only transferred back to the host after the end of the time iteration loop.

Having considered the parallel design implications, the remaining sections detail the different versions of the timpani drum that are implemented for the performance analysis.

7.3.2 CSC format using CSparse

The initial implementation in single thread C code on the CPU makes use of the CSparse library [117]. This is a compact, robust, C library that make use of the CSC sparse matrix format. The library contains all of the basic linear algebra functions, as well as direct solvers for sparse linear systems (parts of this library are used to implement the backslash operator in MATLAB).

For the time iteration operations, only a matrix by vector multiplication function is needed (dot products are trivial to compute with a loop). The CSparse library provides a *gaxpy* function to compute this, which also sums into the result any data already held in the output vector $\mathbf{y}$. For the timpani time iteration loop, this does require initialising

some vectors to zero before they are used. Note that complex operations, such as those found in Algorithm 17 for calculating the membrane, necessitate multiple *gaxpy* calls and storage of the results in temporary vectors.

The CSparse library contains only the CSC format, and does not perform multi-threaded operations.

7.3.3 CSR format and CUSPARSE

Having created a CSC implementation, it is then straightforward to switch to one using the CSR format. Converting between the two can be achieved by taking the transpose of a matrix, and then switching the meaning of the data pointers. The purpose of testing a CSR format is twofold.

Firstly, to test a custom *gaxpy* function using pthreads to perform the operation over multiple cores of the CPU. Parallelisation of matrix by vector multiplication makes use of the row independence of the operations, and so the CSR format is more efficient than the CSC format for this purpose.

Secondly, as the matrices can then be used to test the CUSPARSE library on the GPU. The matrix by vector multiplication function from CUSPARSE, alongside the dot product from the CUBLAS library, are all that is required to implement the time iteration loop.

The *gaxpy* function in C code is implemented to allow two different modes of operation; either summing existing data in the output vector, or overwriting it. This allows a small optimisation over the previous CSC version, as performing a separate initialisation of vectors is not required.

7.3.4 DIA and ELLPACK format

The matrices used in the time iteration loop do not all conform to the strict diagonal banding structure best suited to the DIA sparse matrix format (Section 6.1.2). The membrane state matrices $\mathbf{B}$, $\mathbf{C}$, and $\mathbf{V}$ can make use of the DIA format, but the interpolation matrices $\mathbf{I}^{32}$ and $\mathbf{I}^{23}$ can not. Whilst these matrices are banded, they are not square, and so the DIA format would be extremely inefficient for storage.

However, the ELLPACK format is ideally suited for this purpose, as there are generally a consistent number of elements on each row of the matrices. The column coordinates are stored separately, and this allows a compact representation that makes use of less memory space than the more general CSR or CSC formats.

7.3.5 Unrolled matrix-free operations

As discussed in Section 6.5, uniform explicit finite difference schemes can be expressed computationally as either a system based on matrix by vector multiplications, or as an individual update equation based on a stencil and using a number of constant coefficients. As parts of the membrane update for the timpani take the same basic form, it is possible to unroll elements of the calculation into a matrix-free system (see Figure 7.9).

The objective is to remove the large-scale data storage required for the matrices, and instead perform more calculations through the direct use of a function. This is a particularly useful strategy for the GPU, where the computations are generally memory bandwidth limited. This can also be applied to the interpolation matrices, although with some limitations.

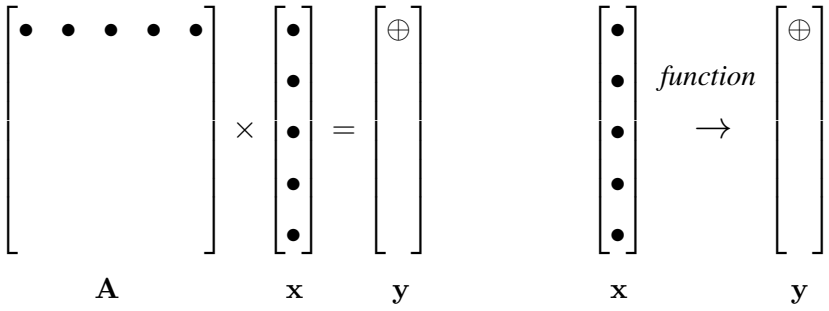


Figure 7.9 *Matrix by vector multiplication (left) can be unrolled into a matrix-free version (right) if a function can be written to generate the result of each row of the multiplication.*

The membrane calculation requires three state matrices; $\mathbf{B}$, $\mathbf{C}$, and $\mathbf{V}$. The five band matrices $\mathbf{C}$ and $\mathbf{V}$ are of the same nearest neighbour type as discussed in Section 6.5.2. However, the matrix $\mathbf{B}$ is a thirteen band matrix which has a more complex structure, but is still block Toeplitz. The stencil representing the matrix behaviour is shown in Figure 7.10.

The membrane is updated at step 3 of Algorithm 17 by applying matrix $\mathbf{B}$ to $\mathbf{u^n}$, and matrix $\mathbf{C}$ to $\mathbf{u^{n-1}}$. When unrolled this becomes an explicit update equation with

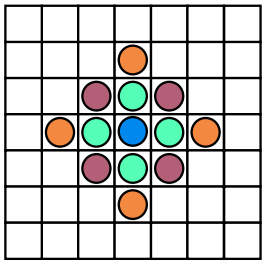


Figure 7.10 *Stencil for the thirteen band matrix operation on* $\mathbf{u^n}$.

the following form:

$$\begin{aligned} u_{l,m}^{n+1} = b_1 u_{l,m}^n + b_2(&u_{l+1,m}^n + u_{l-1,m}^n + u_{l,m+1}^n + u_{l,m-1}^n) \\ + b_3(&u_{l+1,m+1}^n + u_{l+1,m-1}^n + u_{l-1,m+1}^n + u_{l-1,m-1}^n) \\ + b_4(&u_{l+2,m}^n + u_{l-2,m}^n + u_{l,m+2}^n + u_{l,m-2}^n) \\ + c_1 u_{l,m}^{n-1} + c_2(&u_{l+1,m}^{n-1} + u_{l-1,m}^{n-1} + u_{l,m+1}^{n-1} + u_{l,m-1}^{n-1}) \end{aligned} \tag{7.9}$$

The update at step 1 using matrix $\mathbf{V}$ is unrolled using the standard nearest neighbour points.

The interpolation matrices do not contain constant, or block constant, coefficients. However, the elements are the result of bilinear interpolation calculations and so in the case of the matrix $\mathbf{I}^{32}$ the elements can be easily computed by a function using the grid sizes and the circular masking operation.

The transpose of this matrix, $\mathbf{I}^{23}$, is a much more complex function to compute. Unlike $\mathbf{I}^{32}$ which has at most four elements on any row, the transpose has many elements that contribute to a row calculation. Unrolling this as a function does not produce an algorithm that can be efficiently implemented on the GPU. Therefore, only the $\mathbf{I}^{32}$ matrix is unrolled for this implementation, and the $\mathbf{I}^{23}$ matrix is left intact.

7.4 Performance analysis

A performance analysis of the timpani model makes a comparison between the different sparse matrix formats and versions with most of the matrices unrolled. In terms of CPU code, four versions are tested:

1. CSC format using CSparse.
2. Custom CSR format.

3. Custom DIA and ELLPACK.
4. Unrolled matrices, using CSR for the remaining.

For the CUDA code running on the GPU, three versions are tested:

1. CSR using the CUSPARSE library.
2. Custom DIA and ELLPACK.
3. Unrolled matrices, using DIA and ELLPACK for the remaining.

The CUDA code is tested on the Tesla K20 GPU, and results for the C code use the Intel i7 processor. A multi-threaded version of the C code was tested, where the 3D air space update and matrix by vector multiplications make use of multiple threads. Whilst this showed some efficiency gains on the Intel Xeon processor, it still did not match the single thread performance on the i7 processor.

Hence the fastest CPU versions were single threads running on the i7, and these are used for benchmarking purposes here. All computation is at double precision floating-point, and host C code is compiled using -O3 optimisation.

7.4.1 Test simulation

A single test case simulation is used, which models a 29inch drum that is tensioned to produce a G2 pitch. This is embedded in a $1m^3$ anechoic air space, and uses the following input parameters.

Parameter	**Value**
Wave speed of air	344m/s
Air density	1.21 kg/m^3
Depth of timpani shell	0.5 m
Membrane density	0.262 kg/m^3
Tension	2200 N/m
Radius	0.37 m
Thickness	0.19 mm
Young's modulus	3e9

Table 7.2 *Test simulation parameters, with physical properties of a Mylar membrane.*

It is computed at a sample rate of 44.1kHz, and for 44,100 time steps, to produce one second of output. The drum is struck once with a high amplitude at the beginning of the simulation, and ten iterations of the linear system solution are used at each time step. Note that this could be lowered, but for the purpose of CPU to GPU comparisons the number of iterations is not relevant. Table 7.3 shows the resulting sizes of the state grids and boundary shell list.

Element	**Dimensions**	**Total points**
3D air space	$75 \times 75 \times 75$	421,875
2D drumhead	170×170	28,900
Air space interaction layer	55×55	3,025
Drum shell boundary list (ten columns)	$12{,}416 \times 10$	124,160

Table 7.3 *Timpani test simulation grid sizes at 44.1kHz.*

7.4.2 Summary comparisons

Table 7.4 shows the computation times for the various versions running on the i7 CPU and K20 GPU, as well as the associated speedup in each case.

Version	**Intel i7 CPU** (seconds)	**K20 GPU** (seconds)	**Speedup**
CSC Csparse	231.5	-	-
CSR/CUSPARSE	204.6	77.3	2.6X
DIA/ELLPACK	389.5	78.6	5.0X
Unrolled matrices	199.2	67.0	3.0X

Table 7.4 *Timpani test results for a one second simulation at 44.1kHz.*

Starting with the CPU results, the DIA/ELLPACK version is considerably less efficient then the other three implementations, taking nearly twice as long to compute. This is as expected, due to the extra calculations required to compute the data positions. The CSR format is the optimal matrix version, and shows significant gains over the basic CSC library version. The unrolled matrix code shows a further small improvement over the CSR, but the gains are minimal.

Moving to the GPU, there is little to choose between the matrix versions. Both the CUSPARSE library, and the DIA/ELLPACK show similar times, despite the reduction in data movement with the optimised format. The matrix-free unrolled version, however, shows a significant improvement. For both the CPU and GPU code, the version using the unrolled matrices produces optimal efficiency.

To better understand these results, the following section gives an analysis of the different stages of a single time step of the simulation.

7.4.3 Analysis of a single iteration in time

The code versions were tested to analyse the execution times for each of the seven stages of a single time iteration. The final iteration of the simulation was used in each case (i.e. time step 44,100). Table 7.5 shows the results from the single thread code running on the Intel i7 processor. Timings are given here in microseconds.

Stage	**CSC format** (μs)	**CSR format** (μs)	**DIA/ELL** (μs)	**Unroll/CSR** (μs)
1. Calculate Laplacian	690	691	689	690
2. Adjust Laplacian	138	137	136	138
3. Calculate drumhead	1,107	949	2,041	699
4. Linear system solution	2,613	2,070	4,657	2,070
5. Update air space	650	648	647	646
6. Adjust 3D layer	158	137	671	137
7. Read output	1	1	1	1
	5,357	4,633	8,842	4,381

Table 7.5 *Timpani analysis for a single time iteration on the i7 CPU.*

Stages 1,2,5 and 7 are equivalent across each implementation as there are no matrix or unrolled operations. Stage 6 represents only a small percentage of the overall time, although note that the DIA/ELLPACK version is considerably slower than the other versions. The major differences occur at stages 3 and 4.

For stage 3, calculating the drumhead membrane, the CSR format is marginally more efficient than the CSC, whilst the DIA/ELLPACK version is twice as slow. The unrolled version gives a small improvement over the best matrix version.

The linear system solution at stage 4 is the major computational component, taking between 45% and 53% of the overall time for the iteration. Here, the CSR format and the unrolled are equally optimal. The unrolling at this stage is applying a matrix-free version of the $\mathbf{I}^{32}$ update, but the volume of calculation required does not lead to efficiency gains over the basic CSR version. The DIA/ELLPACK form is over twice as slow as the CSR format on the CPU. As mentioned in section 6.4.2, an OpenMP threading approach may obtain further efficiency gains over these single thread results, depending on the size of the matrices being used.

Table 7.6 shows the results for the GPU versions running on the Tesla K20. The CSR format is using the CUSPARSE libray.

Stage	**CSR format** (μs)	**DIA/ELL** (μs)	**Unroll/DIA/ELL** (μs)
1. Calculate Laplacian	58	59	57
2. Adjust Laplacian	13	14	13
3. Calculate drumhead	318	273	210
4. Linear system solution	1,019	1,150	1,002
5. Update air space	91	88	93
6. Adjust 3D layer	76	75	76
7. Read output	4	4	4
	1,579	1,663	1,455

Table 7.6 *Timpani analysis for a single time iteration on the K20 GPU.*

For the stage 3 calculation of the drumhead membrane, the CSR format is now the least efficient version. The DIA/ELLPACK gives a small improvement, and the unrolled version is significantly faster. However, this stage is takes only around one third of the time of stage 4.

For the linear system solution, the DIA/ELLPACK version is slightly less efficient than the CSR, most likely due to the small size of the $\mathbf{I}^{23}$ matrix (it has only 3,025 rows). Issuing only three thousand threads to perform the matrix by vector operation is not sufficient to obtain significant performance gains. The matrix operations in the linear system solution involve only ELLPACK format matrices, and in this instance there is no benefit over the CSR form.

Again the unrolled version is optimal, clearly demonstrating that reducing memory access requirements is the best strategy for overall performance optimisation. The ELLPACK matrix is still used in this unrolled version, as it allows a custom function to be written that combines the operations at step 2 of Algorithm 18. Using the CUSPARSE library requires three separate function calls, but this can be reduced to two using a custom function.

As a final comparison, Table 7.7 shows the speedups obtained for each stage for the fastest GPU version over the fastest CPU version (which is the unrolled form in each case).

Stage	**Intel i7 CPU** (μs)	**K20 GPU** (μs)	**Speedup**
1. Calculate Laplacian	690	57	12.1X
2. Adjust Laplacian	138	13	10.6X
3. Calculate drumhead	699	210	3.3X
4. Linear system solution	2,070	1,002	2.1X
5. Update air space	646	93	7.2X
6. Adjust 3D layer	137	76	1.8X
7. Read output	1	4	0.3X
	4,381	1,455	3.0X

Table 7.7 *Single iteration analysis of fastest Intel i7 CPU versus fastest Tesla K20 GPU versions.*

The stages that involve updating the three-dimensional data (1, 2 and 5) achieve the greatest speedups, as is to be expected where a large number of threads can be issued. The complex stage 3 drumhead calculation obtains a more modest 3.3X performance gain. However, the limiting factor is the linear system solution which only produces 2X acceleration, and is the main reason for the figure of 3X for the overall speedup. Ultimately, the small row size of the $\mathbf{I}^{23}$ matrix and the dot product calculation, and the repeated processing of these inside the iterations of the linear system solution, bound the available performance benefit from the GPU.

Having examined the computational optimisation of the single instrument model, this can then be used as the basis for a large-scale, multiple instrument simulation.

7.5 A multi-instrument model in a virtual room

The model of the timpani drum enclosed in a small anechoic space is useful for testing purposes in order to understand the range of the input parameters, and to optimise the computational elements. However, producing more realistic simulations requires placing the model inside larger scale acoustic settings such as those described in Chapter 5. Even using a single GPU card such as the Tesla K20, a volume of a 500 cubic metres can be modelled using all available memory, at 44.1kHz and double precision, using a three grid scheme (see Section 5.4). This is of a suitable scale that multiple timpani can be placed inside the acoustic environment, and played simultaneously. Using four GPU cards together, it is possible to simulate spaces that are close to a small concert hall in size, all at a sample rate of 44.1kHz.

7.5.1 System abstraction

To create simulations using multiple timpani placed inside such spaces, we first need to abstract the computational components for the drum and represent the elements as objects which can be created and placed inside any given acoustic space. From an algorithm perspective, this requires de-coupling the elements of the drum that use the three-dimensional air space. Specifically, this means refactoring the calculations involving the Laplacian operator such that the intermediate storage is no longer necessary. The previous algorithm required two passes over the three-dimensional data, which is clearly inefficient when dealing with a large-scale data set.

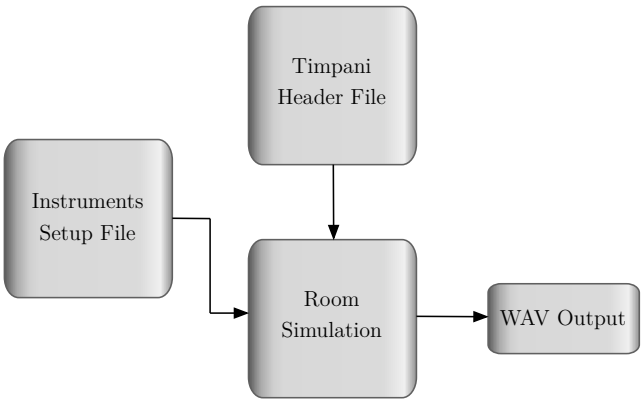


Figure 7.11 *Multiple timpani model system setup.*

The elements of the timpani model can then be described in some encapsulated form, either as a C++ class, or in C using suitable structs and functions. A C code implementation is used here, with a header file that presents a simple interface in the form of `createTimpani()`, `processTimpani()`, and `deleteTimpani()` functions. A complete system can then be created that allows any number of timpani drums to be played inside a room simulation (Figure 7.11). An instrument setup file defines the initial parameters for each drum, as well as the definitions of the strikes to each drumhead. This can be stored in a similar manner to a CSound orchestra and score files [164].

7.5.2 Example simulation

Figure 7.12 shows a snapshot of a layer of a simulation where four timpani are placed inside a room simulation. The drums range in size from 23 to 32 inches in diameter, representing the standard concert set. Various audio examples of this system are available on the accompanying DVD media.

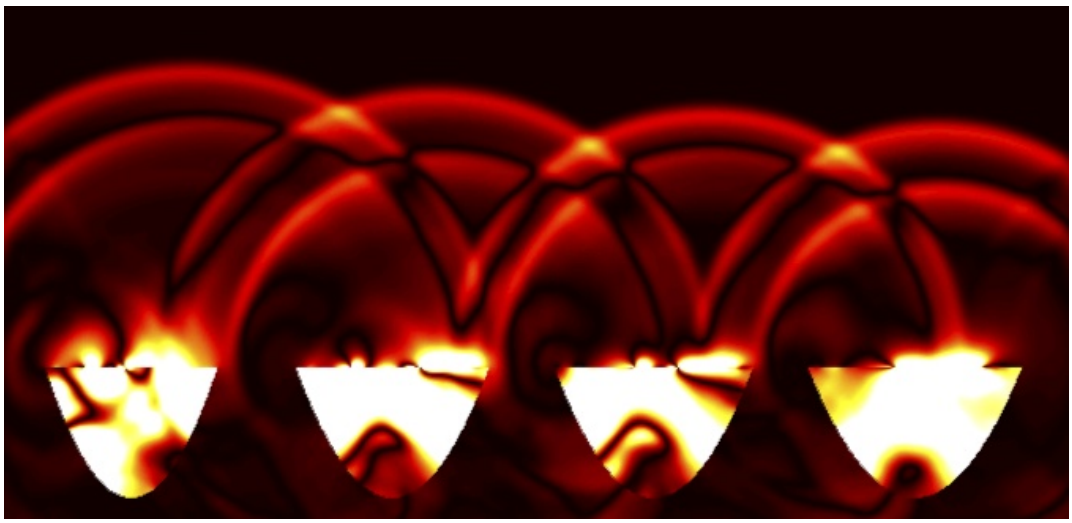


Figure 7.12 *Snapshot of a 3D room simulation with four timpani drums embedded in the space.*

When using a large-scale simulation space at some point the computation of the three-dimensional air space will dominate the simulation time compared to the calculations of the timpani drum. The point at which the room takes longer to compute than a single timpani can be approximated.

From Chapter 4, the standard basic scheme at 44.1kHz and double precision takes 155 seconds to compute 44,100 time steps for a $38m^3$ space. This gives a timing of 3,515ms per time step. Therefore, assuming that computation time scales linearly with volume at this level of computation, a $15.5m^3$ space would take 1,450ms, approximately the same as the test simulation for the timpani drum. So, for any type of large-scale concert space, the time taken to compute a timpani drum update will be only a small percentage compared to the remaining calculations.

7.5.3 Further parallel implementations

As a further consideration, the use of multiple timpani drums could also benefit from a high level parallel design. With the previous simulation, the four timpani drums were computed in a sequence in a single CUDA stream. So the time loop stages for timpani number one are performed, followed by the stages for number two, as so forth.

A high level parallel optimisation when using multiple GPU devices would be to allocate an individual instrument to each device. In that manner each drum could be updated simultaneously, and with the additional benefit of distributing the memory requirements over the multiple GPUs.

7.6 Summary

This chapter has demonstrated the level of acceleration achievable with a complex model of a instrument. The overall performance gain of 3X for the single timpani drum simulation is largely in line with the results from Chapters 6, given the dominant effect of the linear system solution on the computation time.

Whilst the DIA matrix storage format provided some improvement over CSR for the main membrane calculations, the ELLPACK format was less successful for performing the matrix by vector multiplications for the linear system solution stage. However, optimal performance was achieved by unrolling many of the matrix operations into direct, matrix-free, operations. This applies to both the CPU (to a lesser extent), and to the GPU where the reduction in memory usage at the cost of extra calculation provides substantial efficiency gains. When considering large-scale simulation spaces such as hundreds or thousands of cubic metres, the level of computation required for this form of instrument is minimal compared to the updating of the three-dimensional air space as a whole.

The membrane system used here makes use of simplifications to the von Kármán method for nonlinear membranes and plates, and these simplifications allow the Jacobi-type solver to be used. In the case of a full von Kármán system, a more complex solver would be required, such as the Preconditioned Conjugate Gradient method as detailed in Section 2.4.3. This has been demonstrated for drum models such as the snare and bass drum [165].

Whilst algorithms such as PCG for sparse matrices are straightforward to compute in a single-threaded CPU environment, obtaining acceleration through the parallel architecture of a GPU is not always possible [133]. The standard use of the PCG algorithm is to perform a triangular solve on the factors of the preconditioner matrix. However, due to the inherently serial nature of the triangular solve method, it is often difficult to obtain efficiency gains on the GPU when dealing with sparse matrices. Some potential for acceleration has been shown [166], but this greatly depends on the size and sparsity patterns of the matrices used in the triangular solve.

In terms of full von Kármán membranes, initial experimental results by the author have not shown any efficiency gains over a standard CPU implementation. Hybrid methods that combine GPU and CPU computation could well be an optimal solution, especially with increases in the memory transfer speeds between host and device that are likely in future GPU hardware.

More advanced auralizations can be created by moving the readout position(s) during the runtime of the simulation. A simple linear, bilinear, or trilinear interpolation can move the readout position through the acoustic field on any chosen path that is provided. In this manner a fully dynamic auralization can be created.

As a further note, it is also possible to attempt binaural simulations with the use of an embedded model of a human or artificial head. Just as the rigid shell of the timpani drum is embedded in the three-dimensional space, a head model can be embedded using the same techniques [167]. However, whilst it is relatively straightforward to perceive horizontal position using such an approach, position in the vertical plane is much harder to reproduce [168].

Part IV

Summary and conclusions

Chapter 8

Summary and conclusions

This thesis has examined optimisation strategies and GPU performance efficiency of FDTD schemes for virtual acoustics and embedded three-dimensional physical models of instruments, all using full audio sample rates such as 44.1kHz. This chapter presents a summary of the main efficiency results and optimisation techniques from Chapter 3 to Chapter 7, as well as the concluding remarks.

8.1 Summary of results

Chapter 3 made use of one single test case simulation, and examined the various options for executing the model in C code on both the Intel i7 and Xeon CPU processors, as well as CUDA code running on the Fermi and Kepler generation Tesla GPU devices. The test simulation is a $38m^3$ space modelled using the most basic reduced form of the second order 3D wave equation scheme, with a zero boundary condition. It is computed at 44.1kHz, and for 44,100 time steps to give a one second simulation. The state data grids each contain 15.7 million points.

Initial single thread CPU codes take nearly two and half hours to complete the simulation, on the Xeon processor. The use of compiler optimisation results in significant gains, close to 4X on both CPU processors. Multi-threaded POSIX code is then tested, and shows the performance potential of these multi-core CPUs. From two and half hours, the best CPU time is now just over six minutes.

There are multiple options for implementing the scheme on a GPU using CUDA, and so a comparison is made between six different kernel code implementations. These use variations of thread models, and approaches to the use of shared memory, and employ cache optimisations such as the read-only data cache of the Kepler architecture.

The differences between the efficiency of these kernels can be up to 1.8X on the Tesla K20 device, although results vary according to the floating-point precision level that is used. Whilst a 2D slicing shared memory approach is optimal in some cases, the use of direct access to global memory with cache optimisation is very close to optimal across all tests.

The use of multiple GPUs in CUDA is then examined, and a comparison is made between non-asynchronous and asynchronous implementations over four GPU devices. An optimised asynchronous approach shows the greatest efficiency, allowing speedups of 3.5X to 3.7X over single device results.

In terms of the overall benchmark performance, the most realistic comparisons between CPU and GPU processors show speedups of 5X at single precision and 3X at double precision floating-point, when comparing the fastest multi-thread CPU version to the fastest single GPU results. This is far from the 100X speedups that have been reported for GPU acceleration, but reflect the impact of proper benchmarking when performing these kinds of comparisons.

Chapter 4 compared the relative GPU performance of alternative FDTD schemes for computing the three-dimensional wave equation, to the performance of the basic scheme from the previous chapter. First the staggered grid form of the basic scheme is compared, in this case using three different volumes of 1m^3, 38m^3, and 250m^3. This scheme runs at 3.4X slower than the benchmark basic form, across both single and double precision, as well as using 50% more memory for the same physical size of simulation.

This was followed by an analysis of two schemes which have different characteristics to the basic 3D scheme, in terms of cutoff frequencies and dispersion behaviour. The 27-point IWB scheme and 13-point FCC scheme are compared, using three sets of tests. These examined GPU performance when using an equal number of grid points for each scheme, an equal physical size, and finally using a normalised computational density. The FCC scheme shows some efficiency gains over the IWB scheme in each case, and in terms of the first and third tests it produces timings close to the basic 3D scheme, running at 1.2X slower even though the scheme uses twice the number of grid points and has a more complex kernel arrangement. The overall differences in the bandwidth requirements make comparisons difficult to judge.

Whilst previous chapters focused purely on computing three-dimensional wave propagation, such models cannot be used for producing virtual acoustics simulations without appropriate boundary conditions. Chapter 5 began by assessing the GPU effi-

ciency and implementation issues that arise when including boundary computation into the simulation. It first compared the performance of a simple frequency-independent lossy condition that does not require additional state memory to be held. It demonstrates that a highly efficient SIMD strategy can be employed that computes both the interior and exterior grid points with minimal conditional statements in the CUDA kernel. This achieves results that are very close to the basic benchmark scheme.

More complex boundary conditions were then examined that necessitate the use of extra state memory stored at the boundary domain. The implementation of such schemes was examined with the use of a second order absorbing condition. The key component in terms of GPU efficiency is maintaining coalesced memory transfers, and an optimised strategy was demonstrated that remaps the faces of the domain that contain non-contiguous memory. This provides significant efficiency gains over an unoptimised version.

Following this, a more advanced three-dimensional scheme was detailed, that also includes a high-frequency damping viscosity element. This requires twice the number of reads from global memory, and 50% more memory, but gives GPU performance of 1.5X slower compared to the basic 3D scheme.

Issues relating to single precision floating-point were also examined, in terms of both the instabilities that can arise, and the accuracy of the output. When computing simulations at the Courant limit, even the simple frequency-independent boundary scheme can become unstable when using single precision. The simulation may begin stable, and only start to drift away to exponential instability after many thousands of time iterations. Backing away from the Courant limit by a small amount eliminates this behaviour. Even when the simulation is stable, differences can develop in the output between a single precision simulation and an equivalent double precision model. These can increase to normalised differences of around 10^{-3}, which is considerably higher than the floating-point accuracy at single precision. Finally, this chapter used a large-scale cube simulation to demonstrate the effect of dispersion in the basic 3D scheme, and its variation according to direction away from the input source.

In order to create three-dimensional physical models of instruments, additional systems such as strings, plates or membranes need to be embedded into the virtual acoustic spaces examined in previous chapters. These systems often make use of, or require, *sparse* linear algebra constructs in their design. Therefore, Chapter 6 examined the CPU and GPU performance of basic vector operations, as well as matrix by vector multiplication using various sparse matrix storage formats.

First, simple vector addition is detailed, comparing multi-threaded CPU code to CUDA code running on the Tesla K20 device. At small vector sizes of 10,000 elements there is little difference between CPU and GPU performance. However, for 100,000 elements and above the GPU shows 5X speedups at double precision, and 9X at single precision. The dot product is then examined, which although straightforward to implement on the CPU, requires a more complex GPU implementation using a binary tree approach making use of shared memory. At small vector sizes, the CPU outperforms the GPU, up to around 50,000 elements in size where parity is achieved. Above this, the GPU shows speedups, up to 6X when using 1 million elements.

Matrix by vector multiplication is first considered on the CPU with single thread implementations using the CSR, DIA, and ELLPACK sparse matrix storage formats. The matrices used for this testing have the same sparsity patterns as the matrices that typically arise in two-dimensional FDTD schemes (i.e. diagonally banded) . The CSR format is most efficient on the CPU, followed by ELLPACK, and then the DIA format. Equivalent testing on the GPU used the Nvidia CUSPARSE library for the CSR format, and custom written DIA and ELLPACK functions. On the Tesla K20 GPU, the DIA format is now considerably more efficiency, by up to 2X over the CSR function from CUSPARSE. The ELLPACK format is somewhere in-between.

The DIA format is then examined for multi-threaded CPU and GPU comparison tests, to evaluate the realistic speedups that are obtainable. The GPU achieves 5X speedups at double precision, and 7X at single precision. Finally, a simple two-dimensional wave equation scheme is used to examine the relative performance of a matrix formation (using matrix by vector multiplication) compared to an 'unrolled' matrix-free version. On the CPU, the matrix-free version is approximately 3X faster than the matrix form. On the Tesla K20 GPU, the matrix-free version is similarly 3X faster than the most efficient matrix implementation.

Having demonstrated the GPU performance of schemes for three-dimensional wave propagation, and also for linear algebra operations, Chapter 7 gives a detailed analysis of a complex embedded system that models a timpani drum. It examined various different implementations of a test case simulation of the model in order to assess the relative CPU and GPU performance in each case. These different implementations make use of the various sparse matrix storage formats detailed in Chapter 6, as well as a version that uses elements that are unrolled into matrix-free operations. For the CPU codes, the CSR format implementation is again the most efficient matrix version, whilst a version using unrolled components achieves a small efficiency gain. The

version using the DIA and ELLPACK formats is nearly twice as slow.

In terms of the GPU, there is little difference between the CSR and DIA/ELLPACK versions, mostly due to the relatively small row size of one of the key components of the linear system solution. Whilst the DIA format allows efficiency gains for computing components of the drum membrane, the ELLPACK format used in the linear system solution limits the available performance gains. The version using the unrolled matrix-free elements achieves optimal efficiency, some 15% faster than the best matrix version. Ultimately a 3X speedup is achieved when comparing the fastest GPU version over the fastest CPU version. For a simulation using a single timpani drum enclosed in a $1m^3$ space, the computation time is just over one minute per second of output at a sample rate of 44.1kHz.

Finally an abstracted instrument system was demonstrated, using an example of four timpani drums embedded in a room simulation. When using large-scale room or hall models over approximately $15m^3$, the computation for the three-dimensional air propagation outweighs the computation for the elements of an individual timpani drum.

8.2 Concluding remarks

The key strategies demonstrated through this thesis that have been shown to give optimal efficiency are as follows.

1. *Computing the basic 3D scheme:* For the basic wave equation scheme a 2D slicing approach that makes use of shared memory to reduce the bandwidth requirements is marginally more efficient. However, a simple 3D tiling approach using cache optimisation produces results that are close to optimal at both single and double precision floating-point.

2. *Use of multiple GPUs:* When using multiple devices with CUDA, an asynchronous approach using dual streams produces speedups that are close to linear, by hiding the latency involved in transferring data across the PCIe bus. The results still depend on the size of the halo data being used, and this should improve with faster PCIe transfer speeds.

3. *Simple state-free boundaries:* A highly efficient single kernel with minimal conditional statements produces results that are very close to the test case zero boundary simulation.

4. *Boundaries requiring extra state:* For more complex boundaries that requires extra state memory to be stored, a remapping approach that facilitates memory coalescing produces significant performance gains over a standard implementation.

5. *Sparse matrix operations:* In terms of CPU performance for diagonally banded matrix by vector multiplication, the CSR format produces the most efficient results. However, on the GPU the DIA format shows up to 2X performance gains over the CSR format. The ELLPACK format also shows some performance benefit, but to a lesser extent. The relative CPU to GPU performance does depend on the size of the vectors and matrices being used.

6. *Matrix-free operations:* Where possible, unrolling a matrix form update into a matrix-free operation gives significant efficiency gains both on the CPU and on the GPU. A strategy of increasing the level of computation whilst reducing the memory bandwidth requirements is particularly useful when applied to complex systems of embedded instruments.

7. *Final GPU performance gains:* The overall GPU performance gains for the systems examined here are in the range of 3X to 5X over representative CPU benchmarks, depending on the floating-point precision level used.

A question that can then be asked is why, given the degree of GPU optimisation used, do we only see 3X to 5X speedups? The answer is that the computations being performed generally have a low compute to memory access ratio, typically in the range of one-to-one, and so they are memory bandwidth limited. If we consider the best double precision results from Chapter 3, the effective memory bandwidth achieved is around the 200GB/s mark, and that is the maximum available on the Tesla K20 GPU. However, the approximate flop rate achieved is only 150 Gflop. This is only 13% of the theoretical peak double precision capability of the K20 GPU, which is over a teraflop. The same applies when considering the single precision results.

Whilst the majority of the computations allow enough threads to be issued to obtain high occupancy rates, they are ultimately limited by the ability to get data onto the registers. The next generation architectures should increase both the memory bandwidth on the device, and have faster access to the host, which should offer the potential for greater performance gains when compared to CPU operation alone. Nvidia's roadmap plans for 1TB/sec bandwidth with the Volta generation devices with stacked DRAM,

and the VNLink high-speed interconnect will greatly improve the transfer rates between devices and hosts. This should offer significant performance increases for both single and multiple device codes as used throughout this thesis.

There are of course further avenues that still require analysis, both in terms of the performance of virtual acoustics simulations as well as embedded physical models. For large-scale virtual acoustics, the efficient implementation of complex geometric domains is of particular interest, especially given the importance of coalesced memory access to the overall performance. A complete model for room acoustics would require the implementation of complex geometries with more advanced frequency-dependent boundaries defined throughout the domain. The use of implicit schemes for the three-dimensional propagation in order to reduce dispersion is a further area to be examined, alongside detailed perception testing and analysis to assess the accuracy of such complex models. Achieving binaural output through the use of embedded head models is also feasible, but again more advanced boundary conditions are necessary to implement the required level of detail without the use of excessively higher sample rates.

For embedded physical models, even small test case simulations where an instrument is enclosed in a 1m^3 air space require over a minute to compute one second of output for the most optimal GPU code. Clearly a further 60X level of acceleration is necessary before the simulation can approach a real time implementation, something that is obviously a desirable goal in the long term for the purpose of sound synthesis. This would require significant improvements in both memory bandwidth, and the latency involved in issuing threads, as the main computations involved are of the order of tens of thousands of independent operations, rather than hundreds of thousands or indeed millions in the case of large-scale virtual acoustics.

A further complication is the use of extended models, for example full von Kármán systems, that require more complex linear system solutions at each time step of the simulation. For the types of sparse matrices that arise, the performance of Krylov subspace methods with preconditioning typically involves some operations that are difficult to parallelise on the many-core architecture of a GPU. In these instances a heterogenous approach that maximises the use of both CPU and GPU together could well provide optimal solutions, especially if CPU and GPU architectures merge in future generations of devices.

References

[1] L. Savioja, *Modeling techniques for virtual acoustics*, Ph.D. thesis, Helsinki University of Technology, Dec. 1999.

[2] M. Vorländer, *Auralization : Fundamentals of Acoustics, Modelling, Simulation, Algorithms and Acoustic Virtual Reality*, Springer-Verlag, Heidelberg, Berlin, first edition, 2008.

[3] M. R. Schroeder and B. F. Logan, "Colorless artificial reverberation," *Journal of the Audio Engineering Society*, vol. 9, no. 3, pp. 192–197, July 1961.

[4] V. Välimäki, J. Parker, L. Savioja, J. Smith, and J. Abel, "Fifty years of artificial reverberation," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 20, no. 5, pp. 1421 –1448, July 2012.

[5] J. H. Rindel, "Computer simulation techniques for acoustical design of rooms," *Acoustics Australia*, vol. 23, no. 3, pp. 81–86, 1995.

[6] S. Goodwin, "3D sound for 3D games - Beyond 5.1," in *Proceedings of the 35th International Conference of the Audio Engineering Society*, London, England, Feb 2009.

[7] Dolby Incorporated, "Dolby Atmos - Next generation audio for cinemas," *[Online]http://www.dolby.com/uploadedFiles/Assets/US/Doc/Professional/*, Feb, 2014.

[8] G. Naylor, "ODEON - Another Hybrid Room Acoustical Model," *Applied Acoustics*, vol. 38, pp. 131–143, 1993.

[9] L. Beranek, *Acoustics*, Acoustical Society of America, Cambridge, Massachusetts, 1993.

[10] P. Morse and U. Ingard, *Theoretical Acoustics*, Princeton University Press, Princeton, New Jersey, 1968.

[11] L.E. Kinsler, *Fundamentals of Acoustics*, New York: Wiley, 4th edition, 2000.

[12] B. Dalenbäck, *A new model for room acoustic prediction and auralization*, Ph.D. thesis, Chalmers Univ. of Tech., Gothenburg, Sweden, 1995.

[13] H. Kuttruff, *Room acoustics*, Applied Science Publishers London, 2d edition, 1979.

[14] A. Krokstad, S. Strom, and S. Sorsdal, "Calculating the acoustical room response by the use of a ray tracing technique," *Journal of Sound and Vibration*, vol. 8, no. 1, pp. 118 – 125, 1968.

[15] N. Rober, U.Kaminski, and M. Masuch, "Ray acoustics using computer graphics technology," in *Proceedings of the 10th International Conference on Digital Audio Effects*. DAFx-07, Bordeaux, France, September 2007.

[16] B M Gibbs and D K Jones, "A simple image method for calculating the distribution of sound pressure levels within an enclosure," *Acustica*, vol. 26, no. 1, pp. 24–32, 1972.

[17] J. Allen and D. Berkley, "Image method for efficiently simulating small-room acoustics," *Journal of the Acoustical Society of America*, vol. 65, no. 4, pp. 943–950, 1979.

[18] J Borish, "Extension of the image model to arbitrary polyhedra," *Journal of the Acoustical Society of America*, vol. 75, no. 6, pp. 1827–1836, 1984.

[19] Odeon, "Room acoustics simulation software," *[Online][Cited: 27th Sept 2013.]http://www.odeon.dk*, Sept, 2013.

[20] T. Funkhouser, "A beam tracing method for interactive architectural acoustics," *Journal of the Acoustical Society of America*, vol. 115, no. 2, pp. 739–756, 2004.

[21] I. A. Drumm, "The Application of Adaptive Beam Tracing and Managed DirectX for the Visualisation and Auralisation of Virtual Environments," in *International Conference on Information Visualisation*, 2005, pp. 961–965.

[22] E. Nosal, M. Hodgson, and I. Ashdown, "Improved algorithms and methods for room sound-field prediction by acoustical radiosity in arbitrary polyhedral rooms," *Journal of the Acoustical Society of America*, vol. 116, no. 2, pp. 970–980, 2004.

[23] S. Siltanen, T. Lokki, S. Kiminki, and L. Savioja, "The room acoustic rendering equation," *The Journal of the Acoustical Society of America*, vol. 122, no. 3, pp. 1624–1635, 2007.

[24] C.J. Webb and S. Bilbao, "Computing room acoustics with CUDA - 3D FDTD schemes with boundary losses and viscosity," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, Prague, Czech, 2011, pp. 317–320.

[25] D. V. Hutton, *Fundamentals of Finite Element Analysis*, McGraw-Hill Science/Engineering/Math, 1 edition, June 2003.

[26] C. A. Felippa, "An appreciation of R. Courant's 'Variational methods of the solution of problems of equilibrium and vibrations'," *Int. Journal of Numerical Methods in Engineering*, vol. 37, pp. 2159–2187, 1994.

[27] R. Cook, Ed., *Concepts and applications of finite element analysis*, John Wiley and Sons, New York, New York, fourth edition, 2002.

[28] A. Quarteroni and A. Valli, *Numerical Approximation of Partial Differential Equations*, Springer Publishing Company Incorporated, 1st ed. 1994. 2nd printing edition, 2008.

[29] A. Pietrzyk and M. Kleiner, "The application of the finite-element method to the prediction of soundfields of small rooms at low frequencies," in *Audio Engineering Society Convention 102*, Mar 1997.

[30] C. Brebbia and R. Ciskowski, Eds., *Boundary Element Methods in Acoustics*, Kluwer Academic Publishers, 1991.

[31] L. Franzoni, D. Bliss, and J. Rouse, "An acoustic boundary element method based on energy and intensity variables for prediction of high-frequency broadband sound fields," *Journal of the Acoustical Society of America*, vol. 110, no. 6, pp. 3071–3080, 2001.

[32] S. Kopuz and N. Lalor, "Analysis of interior acoustic fields using the finite element method and the boundary element method," *Applied Acoustics*, vol. 45, no. 3, pp. 193 – 210, 1995.

[33] S. van Duyne and J. O. Smith III, "Physical modelling with the 2D digital waveguide mesh," in *Proceedings of the International Computer Music Conference*, Tokyo, Japan, September 1993, pp. 40–47.

[34] L. Savioja, T. Rinne, and T. Takala, "Simulation of room acoustics with a 3-D finite-difference mesh," in *Proceedings of the International Computer Music Conference*, Århus, Denmark, September 1994, pp. 463–466.

[35] S. Bilbao, *Numerical Sound Synthesis: Finite Difference Schemes and Simulation*, Wiley, 2009.

[36] S. Bilbao, "Digital waveguide networks in inhomogeneous media," in *Proceedings of the COST G-6 Conference on Digital Audio Effects*, Verona, Italy, December 2000, pp. 249–253.

[37] D. Murphy and D. Howard, "2-D digital waveguide mesh topologies in room acoustic modelling," in *Proceedings of the COST G-6 Conference on Digital Audio Effects*, Verona, Italy, December 2000, pp. 211–216.

[38] L. Savioja and V. Välimäki, "Interpolated 3-D digital waveguide mesh with frequency warping," in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, Salt Lake City, Utah, May 2001, pp. 3345–3348.

[39] D. Murphy, M. Beeson, S. Sheeley, and A. Moore, "Hybrid room impulse response synthesis in digital waveguide mesh," in *Proceedings of the 11th International Conference on Digital Audio Effects*, Espoo, Finland, September 2008.

[40] M. Beeson and D. Murphy, "Roomweaver: A digital waveguide mesh based room acoustics research tool," in *Proceedings of the 7th International Conference on Digital Audio Effects*, Naples, Italy, October 2004, pp. 268–273.

[41] R. Courant, K. Friedrichs, and H. Lewy, "Über die partiellen Differenzengleichungen de mathematischen Physik," *Mathematische Annalen*, vol. 100, pp. 32–74, 1928.

[42] J. Strikwerda, *Finite Difference Schemes and Partial Differential Equations*, Wadsworth and Brooks/Cole Advanced Books and Software, Pacific Grove, California, 1989.

[43] A. Taflove, *Computational Electrodynamics*, Artech House, Boston, Massachusetts, 1995.

[44] R. Coates and M. Schoenberg, "Finite difference modeling of faults and fractures," *Geophysics*, vol. 60, no. 5, pp. 1514–1526, 1995.

[45] R.M. Alford, K.R. Kelly, and D.M. Boore, "Accuracy of finite-difference modeling of the acoustic wave equation," *Geophysics*, vol. 39, no. 6, pp. 834–842, 1974.

[46] D. Botteldooren, "Finite-difference time-domain simulation of low-frequency room acoustic problems," *Journal of the Acoustical Society of America*, vol. 98, no. 6, pp. 3302–3308, 1995.

[47] R. Leveque, *Finite Volume Methods for Hyperbolic Problems*, Campbridge University Press, Cambridge, UK, 2002.

[48] S. Bilbao, "Modeling of complex geometries and boundary conditions in finite difference/finite volume time domain room acoustics simulation," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 21, no. 7, pp. 1524–1533, 2013.

[49] J. Botts and L. Savioja, "Integrating finite difference schemes for scalar and vector wave equations," in *IEEE International Conference Acoustics, Speech and Signal Processing*, Vancouver, BC, Canada, 2013, pp. 171–175.

[50] S. Bilbao and J. O. Smith III, "Finite difference schemes and digital waveguide networks for the wave equation: Stability, passivity and numerical dispersion," *IEEE Transactions on Speech and Audio Processing*, vol. 11, no. 3, pp. 255–266, 2003.

[51] A. Southern, D.T. Murphy, T. Lokki, and L. Savioja, "The perceptual effects of dispersion error on room acoustic model auralization," in *Proceedings of Forum Acusticum, Aalborg, Denmark, June 27 - July 1*, 2011, pp. 1553–1558.

[52] K. Kowalczyk, *Boundary and medium modelling using compact finite difference schemes in simulations of room acoustics for audio and architectural design applications*, Ph.D. thesis, School of Electronics, Queen's University Belfast, 2008.

[53] J.S. Juntunen and T.D. Tsiboukis, "Reduction of numerical dispersion in FDTD method through artificial anisotropy," *Microwave Theory and Techniques, IEEE Transactions on*, vol. 48, no. 4, pp. 582–588, 2000.

[54] D. Howard and J. Angus, *Acoustics and Psychoacoustics*, Butterworth-Heinemann, Newton, MA, USA, 2nd edition, 2000.

[55] J. Berenger, "Three-dimensional perfectly matched layer for the absorption of electromagnetic waves," *Journal of Computational Physics*, vol. 127, no. 2, pp. 363–379, September 1996.

[56] K. Kowalczyk and M. van Walstijn, "Virtual room acoustics using finite difference methods. How to model and analyse frequency-dependent boundaries?," in *Communications, Control and Signal Processing, 2008. ISCCSP 2008. 3rd International Symposium on*, march 2008, pp. 1504 –1509.

[57] M. Ament, R. Nothen, M. Vorländer, and D Schroder, "Combined broadband impulse responses using FEM and hybrid ray-based methods," in *Proceedings of the EAA Auralization Symposium*, 2010, pp. 583–592.

[58] A. Southern, S. Siltanen, D.T. Murphy, and L. Savioja, "Room impulse response synthesis and validation using a hybrid acoustic model," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 21, no. 9, pp. 1940–1952, 2013.

[59] N. Raghuvanshi, N. Galoppo, and M. Lin, "Accelerated wave-based acoustics simulation," in *Proceedings of the 2008 ACM symposium on Solid and physical modeling*, New York, NY, USA, 2008, SPM '08, pp. 91–102, ACM.

[60] R. Mehra, N. Raghuvanshi, L. Savioja, M. Lin, and D. Manocha, "An efficient GPU-based time domain solver for the acoustic wave equation," *Applied Acoustics*, vol. 73, no. 2, pp. 83–94, 2012.

[61] V. Välimäki, J. Pakarinen, E. Cumhur, and M. Karjalainen, "Discrete-time modelling of musical instruments," *Reports on Progress in Physics*, vol. 69, no. 1, pp. 1–78, Jan. 2006.

[62] S. Bilbao, B. Hamilton, A. Torin, C.J. Webb, P. Graham, A. Gray, K. Kavoussanakis, and J. Perry, "Large Scale Physical Modeling Sound Synthesis," in *Proceedings of the Stockholm Music Acoustics Conference*, Stockholm, Sweden, 2013, pp. 593–600.

[63] J. Chowning, "The synthesis of complex audio spectra by means of frequency modulation," *Journal of the Audio Engineering Society*, vol. 21, no. 7, pp. 526–534, 1973.

[64] V. Välimäki and T. Takala, "Virtual musical instruments: natural sound using physical models," *Org. Sound*, vol. 1, no. 2, pp. 75–86, Aug. 1996.

[65] J. Kelly and C. Lochbaum, "Speech synthesis," in *Proceedings of the Fourth International Congress on Acoustics*, Copenhagen, Denmark, 1962, pp. 1–4, Paper G42.

[66] P. Ruiz, "A technique for simulating the vibrations of strings with a digital computer," M.S. thesis, University of Illinois, 1969.

[67] L. Hiller and P. Ruiz, "Synthesizing musical sounds by solving the wave equation for vibrating objects: Part I," *Journal of the Audio Engineering Society*, vol. 19, no. 6, pp. 462–470, 1971.

[68] C. Cadoz, A. Luciani, and J.-L. Florens, "Cordis-anima: A modeling and simulation system for sound and image synthesis," *Computer Music Journal*, vol. 17, no. 1, pp. 19–29, 1993.

[69] J. Adrien, "The missing link: Modal synthesis," in *Representations of Musical Signals*, G. DePoli, A. Picialli, and C. Roads, Eds., pp. 269–297. MIT Press, Cambridge, Massachusetts, 1991.

[70] P. Cook, "Tbone: An interactive waveguide brass instrument synthesis workbench for the NeXT machine," in *Proceedings of the International Computer Music Conference*, Montreal, Canada, 1991, pp. 297–299.

[71] M. Karjalainen, V. Välimäki, and T. Tolonen, "Plucked-string synthesis: From the Karplus-Strong algorithm to digital waveguides and beyond," *Computer Music Journal*, vol. 22, no. 3, pp. 17–32, 1998.

[72] J. O. Smith, "Virtual acoustic musical instruments: Review and update," *Journal of New Music Research*, vol. 33, no. 3, pp. 283–304, 2004.

[73] M. Karjalainen and C. Erkut, "Digital waveguides vs. finite difference schemes: Equivalence and mixed modeling," *EURASIP Journal on Applied Signal Processing*, vol. 7, pp. 978–989, 2004.

[74] A. Chaigne, "On the use of finite differences for musical synthesis. Application to plucked stringed instruments," *Journal d'Acoustique*, vol. 5, no. 2, pp. 181–211, 1992.

[75] C. Bruyns, "Modal synthesis for arbitrarily shaped objects," *Computer Music Journal*, vol. 30, no. 3, pp. 22–37, 2006.

[76] M. Ducceschi, C. Touze, and S. Bilbao, "Sound synthesis of gongs obtained from nonlinear thin plates vibrations: Comparison between a modal approach and a finite difference scheme," in *Proceedings of the Stockholm Music Acoustics Conference*, Stockholm, Sweden, 2013.

[77] J. Bensoam, N. Misdariis, C. Vergez, and R. Caussé, "Musical application with Modalys sound synthesis program based in modal representation," in *Systemics, Cybernetics and Informatics*, Orlando, Florida, 2001.

[78] B. Bank, F. Avanzini, G. Borin, G. De Poli, F. Fontana, and D. Rocchesso, "Physically informed signal processing methods for piano sound synthesis: A research overview," *EURASIP Journal of Applied Signal Processing*, vol. 2003, pp. 941–952, Jan. 2003.

[79] K. Karplus and A. Strong, "Digital synthesis of plucked-string and drum timbres," *Computer Music Journal*, vol. 7, no. 2, pp. 43–55, 1983.

[80] D. Jaffe and J. O. Smith III, "Extensions of the Karplus-Strong plucked string algorithm," *Computer Music Journal*, vol. 7, no. 2, pp. 56–68, 1983.

[81] J. O. Smith III, "Physical modelling using digital waveguides," *Computer Music Journal*, vol. 16, no. 4, pp. 74–91, 1992.

[82] G. Borin, G. DePoli, and A. Sarti, "Algorithms and structures for synthesis using physical models," *Computer Music Journal*, vol. 16, no. 4, pp. 30–42, 1992.

[83] E. Becache, A. Chaigne, G. Derveaux, and P. Joly, "Numerical simulation of a guitar," *Computers & structures*, pp. 1–30, 2005.

[84] N. Giordano and M. Jiang, "Physical modeling of the piano," *EURASIP Journal on Applied Signal Processing*, vol. 2004, no. 1, pp. 926–933, 2004.

[85] R. Bader, *Computational Mechanics of the Classical Guitar*, Springer-Verlag, Berlin Heidelberg, 2005.

[86] S. Bilbao, "A family of conservative finite difference schemes for the dynamical von Karman plate equations," *Numerical Methods for Partial Differential Equations*, vol. 24, no. 1, pp. 193–216, 2008.

[87] B. Gaster, L. Howes, D. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL*, Morgan Kaufmann, 2nd edition, 2012.

[88] G. Henry, B. Cole, P. Fay, and T.G. Mattson, "The performance of the INTEL TFLOPS supercomputer," *Intel Technology journal*, vol. Q1 issue, 1998.

[89] J. Tolke and M. Krafczyk, "TeraFLOP computing on a desktop PC with GPUs for 3D CFD," *International Journal of Computational Fluid Dynamics*, vol. 22, no. 7, pp. 443–456, Aug. 2008.

[90] Nvidia Corp, "CUDA C Programming Guide," *[Online][Cited: 6th Jan 2014.]http://docs.nvidia.com/cuda/index.html*, Sep 2014.

[91] G. Almasi and A. Gottlieb, *Highly parallel computing*, Benjamin Cummings, Redwood City, CA, 1989.

[92] R. Trobec, M. Vajtersic, and P. Zinterhof, *Parallel Computing: Numerics, Applications, and Trends*, Springer Publishing Company, Incorporated, 1st edition, 2009.

[93] G. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the joint computer conference*, New York, NY, USA, 1967, AFIPS '67 (Spring), pp. 483–485, ACM.

[94] J. Gustafson, "Reevaluating Amdahl's law," *Communications of ACM*, vol. 31, no. 5, pp. 532–533, May 1988.

[95] M. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computing*, vol. 21, no. 9, pp. 948–960, Sept. 1972.

[96] C. Trendall and J. Stewart, "General calculations using graphics hardware with applications to interactive caustics," in *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, London, UK, UK, 2000, pp. 287–298, Springer-Verlag.

[97] D. Kirk and W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[98] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: stream computing on graphics hardware," *ACM Transactions on Graphics*, vol. 23, no. 3, pp. 777–786, Aug. 2004.

[99] R. Farber, *CUDA Application Design and Development*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[100] Nvidia Corp, "Fermi Compute Architecture whitepaper," *[Online][Cited: 6th Jan 2014.]http://developer.nvidia.com*, Oct, 2010.

[101] L. Ewijk, "The finite difference time domain method on a massively parallel computer," in *High-Performance Computing and Networking*, vol. 1067 of *Lecture Notes in Computer Science*, pp. 593–598. Springer Berlin Heidelberg, 1996.

[102] C. Fackler, J. Botts, and N. Xiang, "Parallelized finite difference time domain room acoustic simulation," *Journal of the Acoustical Society of America*, vol. 132, no. 3, pp. 1880, 2012.

[103] A. Balevic, L. Rockstroh, A. Tausendfreund, S. Patzelt, G. Goch, and S. Simon, "Accelerating Simulations of Light Scattering Based on Finite-Difference Time-Domain Method with General Purpose GPUs," in *CSE 11th IEEE International Conference on Computational Science and Engineering*, 2008, pp. 327–334.

[104] D. Brandao, M. Zamith, E. Clua, and A. Montenegro, "Performance evaluation of optimized implementations of finite difference method for wave propagation problems on GPU architecture," in *Proceedings of the 22nd International Symposium on Computer Architecture and HPC*. Brazil, 2010.

[105] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units - GPGPU-2*, New York, New York, USA, 2009, pp. 79–84, ACM Press.

[106] K. Hou, Y. Zhao, J. Huang, and L. Zhang, "Performance evaluation of the Three-Dimensional Finite-Difference Time-Domain(FDTD) method on Fermi architecture GPUs," in *Algorithms and Architectures for Parallel Processing*, vol. 7016 of *Lecture Notes in Computer Science*, pp. 460–469. Springer Berlin / Heidelberg, 2011.

[107] A. Nguyen, N. Satish, J. Chhugani, Changkyu Kim, and P. Dubey, "3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs," in *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, 2010, pp. 1–13.

[108] L. Savioja, "Real-time 3D finite-difference time-domain simulations of low and mid-frequency room acoustics," in *13th International Conference on Digital Audio Effects*, Sept, 2010.

[109] J. Sheaffer and B. Fazenda, "FDTD/K-DWM simulation of 3d room acoustics on general purpose graphics hardware using compute unified device architecture," in *Proc. Inst. Acoustics*, 2010, vol. 32.

[110] P. Micikevicius, "Multi-GPU Programming - Nvidia Cuda webinars," *[Online][Cited: 6th Jan 2013.] http://developer.download.nvidia.com*, 2011.

[111] Q. Zhang, L. Ye, and Z. Pan, "Physically-Based Sound Synthesis on GPUs," in *Entertainment Computing - ICEC 2005*, Fumio Kishino, Yoshifumi Kitamura, Hirokazu Kato, and Noriko Nagata, Eds., vol. 3711 of *Lecture Notes in Computer Science*, pp. 328–333. Springer Berlin Heidelberg, 2005.

[112] L. Savioja, V. Välimäki, and J. O. Smith III, "Real-time additive synthesis with one million sinusoids using a GPU," in *Proceedings of the 128th AES Convention*, London, UK, 2010.

[113] M. Sosnick and W. Hsu, "Implementing a Finite DIfference-Based Realtime Sound Synthesizer using GPUs," in *Proceedings of the International Conference on New Interfaces for Musical Expression*, Oslo, Norway, 2011, pp. 264–267.

[114] W. Jiang N. Tsingos and I. Williams, "Using programmable graphics hardware for acoustics and audio rendering," *Journal of the Audio Engineering Society*, vol. 59, no. 9, pp. 628–646, 2011.

[115] L. Savioja, V. Välimäki, and J.O. Smith, "Audio signal processing using graphics processing units," *Journal of the Audio Engineering Society*, vol. 59, no. 1/2, pp. 3–19, Feb 2011.

[116] G. Golub and C. Van Loan, *Matrix computations (3rd ed.)*, Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[117] T. Davis, *Direct Methods for Sparse Linear Systems*, SIAM, 2006.

[118] G. Strang, *Computational Science and Engineering*, Wellesley-Cambridge Press, Wellesley, Massachusetts, 2007.

[119] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery, *Numerical recipes in C (2nd ed.): the art of scientific computing*, Cambridge University Press, New York, NY, USA, 1992.

[120] Y. Saad, *Iterative Methods for Sparse Linear Systems*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[121] M. Benzi and T. Miroslav, "A comparative study of sparse approximate inverse preconditioners," *Applied Numerical Mathematics*, vol. 30, no. 2030706, pp. 305–340, 1999.

[122] N. Bell and M. Garland, "Cusp: Generic parallel algorithms for sparse matrix and graph computations," 2013, Version 0.4.0.

[123] J. Humphrey, D. Price, K. Spagnoli, A. Paolini, and E. Kelmelis, "CULA: Hybrid GPU Accelerated Linear Algebra Routines," in *SPIE Defense and Security Symposium (DSS)*, April 2010.

[124] J. Godwin, J. Holewinski, and P. Sadayappan, "High-performance sparse matrix-vector multiplication on GPUs for structured grid computations," in *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, New York, NY, USA, 2012, GPGPU-5, pp. 47–56, ACM.

[125] D. Grewe and A. Lokhmotov, "Automatically generating and tuning GPU code for sparse matrix-vector multiplication from a high-level representation," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, New York, NY, USA, 2011, GPGPU-4, pp. 12:1–12:8, ACM.

[126] I. Reguly and M. Giles, "Efficient sparse matrix-vector multiplication on cache-based GPUs," in *Innovative Parallel Computing (InPar), 2012*, 2012, pp. 1–12.

[127] D. Mukunoki and D. Takahashi, "Optimization of Sparse Matrix-Vector Multiplication for CRS Format on NVIDIA Kepler Architecture GPUs," in *Computational Science and Its Applications ICCSA 2013*, vol. 7975 of *Lecture Notes in Computer Science*, pp. 211–223. Springer Berlin Heidelberg, 2013.

[128] A. Gupta and V. Kumar, "Parallel algorithms for forward and back substitution in direct solution of sparse linear systems," in *Supercomputing, 1995. Proceedings of the IEEE/ACM SC95 Conference*, 1995, pp. 74–74.

[129] E. Ortigosa, L. Romero, and J. Ramos, "Parallel scheduling of the PCG method for banded matrices rising from fdm/fem," *Journal of Parallel and Distributed Computing*, vol. 63, no. 12, pp. 1243 – 1256, 2003.

[130] D. Egloff, "High performance finite difference PDE solvers on GPUs," Tech. Rep., Technical Report, QuantAlea Gmbh, 2010.

[131] M. Naumaov, "Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU," *Nvidia Technical Report*, 2010.

[132] Y. Aksari and H. Artuner, "Forward and back substitution algorithms on GPU: a case study on modified incomplete cholesky preconditioner for three-dimensional finite difference method," *The Journal of Supercomputing*, vol. 62, no. 1, pp. 550–572, 2012.

[133] R. Li and Y. Saad, "GPU-accelerated preconditioned iterative linear solvers," *The Journal of Supercomputing*, vol. 63, no. 2, pp. 443–466, Oct. 2012.

[134] M. Ament, G. Knittel, D. Weiskopf, and W. Strasser, "A parallel preconditioned conjugate gradient solver for the poisson problem on a multi-GPU platform," in *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, 2010, pp. 583–592.

[135] G. Gravvanis, C. Filelis-Papadopoulos, and K. Giannoutakis, "Solving finite difference linear systems on GPUs: CUDA based parallel explicit preconditioned biconjugate conjugate gradient type methods," *The Journal of Supercomputing*, vol. 61, no. 3, pp. 590–604, 2012.

[136] Nvidia Corp, "CUDA C best practices guide," *[Online][Cited: 6th Feb 2014.]http://docs.nvidia.com/cuda/index.html*, Sep 2014.

[137] A. Southern, *The Synthesis and Auralisation of Physically Modelled Soundfields*, Ph.D. thesis, University of York, 2011.

[138] D. Butenhof, *Programming with POSIX threads*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[139] J. Lopez, D. Carnicero, N. Ferrando, and J. Escolano, "Parallelization of the finite-difference time-domain method for room acoustics modelling based on CUDA," *Mathematical and Computer Modelling*, vol. 57, no. 78, pp. 1822 – 1831, 2012.

[140] C.J. Webb and A. Gray, "Large-scale Virtual Acoustics Simulation at Audio Rates Using Three Dimensional Finite Difference Time Domain and Multiple GPUs," in *Proceedings of Meetings on Acoustics: International Congress on Acoustics*, Montreal, Canada, 2013, vol. 19, p. 70092.

[141] T. Chu, J. Dai, D. Qian, W. Fang, and Yi Y. Liu, "A Novel Scheme for High Performance Finite-Difference Time-Domain (FDTD) Computations Based on GPU," in *Algorithms and Architectures for Parallel Processing*, C. Hsu, L. Yang, J. Park, and S. Yeo, Eds., vol. 6081 of *Lecture Notes in Computer Science*, pp. 441–453. Springer Berlin Heidelberg, 2010.

[142] D. Michea and D. Komatitsch, "Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards," *Geophysical Journal International*, vol. 182, pp. 389–402, July 2010.

[143] J. Lopez, J. Navarro, D. Carnicero, and J. Escolano, "Some comments about graphic processing unit (GPU) architectures applied to finite-difference time-domain (FDTD) room acoustics simulation. Present and future trends," *Proceedings of Meetings on Acoustics*, vol. 19, no. 1, 2013.

[144] V. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 451–460, June 2010.

[145] S. Bilbao, "Optimized FDTD Schemes for 3-D Acoustic Wave Propagation," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 20, no. 5, pp. 1658–1663, 2012.

[146] B. Hamilton and S. Bilbao, "On finite difference schemes for the 3-D wave equation using non-Cartesian grids," in *Proceedings of the Stockholm Music Acoustics Conference (SMAC)*, 2013, vol. 2, pp. 1–8.

[147] B. Hamilton and C.J. Webb, "Room acoustics modelling using GPU-accelerated finite difference and finite volume methods on a face-centered cubic grid," in *Proceedings of the 16th International Conference on Digital Audio Effects*, Maynooth, Ireland, 2013, pp. 336–343.

[148] L. Savioja and V. Välimäki, "Interpolated rectangular 3-D digital waveguide mesh algorithms with frequency warping," *IEEE Transactions on Speech and Audio Processing*, vol. 11, pp. 783–790, 2003.

[149] K. Kowalczyk and M. van Walstijn, "Room Acoustics Simulation Using 3-D Compact Explicit FDTD Schemes," *Audio, Speech, and Language Processing, IEEE Transactions on*, vol. 19, no. 1, pp. 34–46, 2011.

[150] K. Kowalczyk and M. van Walstijn, "A comparison of nonstaggered compact FDTD schemes for the 3D wave equation," in *IEEE International Conference on Acoustics Speech and Signal Processing*. ICASSP, Mar 2010, pp. 197–200.

[151] K. Petkov, F. Qiu, Z. Fan, A. Kaufman, and K. Mueller, "Efficient LBM Visual Simulation on Face-Centered Cubic Lattices," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 5, pp. 802–814, 2009.

[152] B. Engquist and A. Majda, "Absorbing boundary conditions for numerical simulation of waves," *Proceedings of the National Academy of Sciences*, vol. 74, no. 5, pp. 1765–1766, 1977.

[153] A. Southern, D. Murphy, and J. Wells, "Rendering walk-through auralisations using wave-based acoustical models," in *Proceedings of the 17th European Signal Processing Conference*. EUSIPCO 09, Glasgow, UK, 2009.

[154] L. Beranek, "Subjective Rank-Orderings and Acoustical Measurements for Fifty-Eight Concert Halls," *Acta Acustica*, vol. 89, pp. 494508, 2003.

[155] D. Sanchez, D. Yuen, Y. Sun, and G. Wright, "Impact of floating-point precision on boundary layer instabilities modeled on fermi GPU," *Research Report UMSI*, 2011.

[156] J. Sheaffer, M. van Walstijn, and B. Fazenda, "A physically-constrained source model for FDTD acoustic simulation," *Proceedings of the 15th International Conference on Digital Audio Effects (DAFx-12)*, 2012.

[157] M. Harris, "Optimizing parallel reduction in CUDA," *Nvidia Technical Paper. Available : http://developer.download.nvidia.com/compute-/cuda/Website/projects/reduction/doc/reduction.pdf*, 2007.

[158] N. Fletcher and T. Rossing, *The Physics of Musical Instruments*, Springer-Verlag, New York, New York, 1991.

[159] H. Berger, "A new approach to the analysis of large deflections of plates," *Journal of Applied Mathematics*, vol. 22, pp. 465–472, 1955.

[160] L. Rhaouti, A. Chaigne, and P. Joly, "Time-domain modeling and numerical simulation of a kettledrum," *Journal of the Acoustical Society of America*, vol. 105, no. 6, pp. 3545–3562, 1999.

[161] F. Avanzini and R. Marogna, "A modular physically based approach to the sound synthesis of membrane percussion instruments," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 18, no. 4, pp. 891–902, May 2010.

[162] S. Bilbao, "Time domain simulation and sound synthesis for the snare drum," *Journal of the Acoustical Society of America*, vol. 131, no. 1, pp. 914–925, 2012.

[163] S. Bilbao and C.J. Webb, "Physical Modeling of Timpani Drums in 3D on GPGPUs," *Journal of the Audio Engineering Society*, vol. 61, no. 10, pp. 737–748, 2013.

[164] R. Boulanger, Ed., *The Csound Book: Perspectives in Software Synthesis, Sound Design, Signal Processing, and Programming*, MIT Press, Cambridge, Massachusetts, 2001.

[165] A. Torin and S. Bilbao, "Numerical Experiments with Non-linear Double Membrane Drums," in *Proceedings of the Stockholm Music Acoustics Conference*, 2013.

[166] M. Naumaov, "Incomplete-LU and Cholesky Preconditioned Iterative Methods Using CUSPARSE and CUBLAS," *Nvidia Technical Report*, 2010.

[167] C.J. Webb and S. Bilbao, "Binaural simulations using audio rate FDTD schemes and CUDA," in *Proceedings of the 15th International Conference on Digital Audio Effects*, York, UK, 2012, pp. 97–100.

[168] J. Sheaffer, C.J. Webb, and B. Fazenda, "Modelling binaural receivers in finite difference simulation of room acoustics," in *Proceedings of Meetings on Acoustics: International Congress on Acoustics*, Montreal, Canada, 2013, p. 15098.

Appendix A

Hardware specifications

	Core i7 3770S	**Xeon E5-2620**
Processor cores	4	6
Max. threads	8	12
Base clock rate	3.1 GHz	2 GHz
Turbo clock rate	3.9 GHz	2.5 GHz
Memory bandwidth	25 GB/s	42 GB/s
Peak double precision	125 Gflops	192 Gflops

Table A.1 *Intel CPU hardware specifications.*

	Tesla C2050	**Tesla K20**
Architecture	Fermi	Kepler
Compute capability	2.0	3.5
CUDA cores	448	2,496
Clock rate	1.15 GHz	706 MHz
Memory bandwidth	144 GB/s	208 GB/s
Peak double precision	515 Gflops	1.17 Tflops
Peak single precision	1.28 Tflops	3.52 Tflops

Table A.2 *Nvidia GPU hardware specifications.*

Appendix B

Code listings

B.1 Single thread C code for test simulation

```
#define ReaL double
#include "CJW_Audio.h"

// Define Grid dims, modulo Thread block
#define Nl 256
#define Nm 296
#define Np 208

#define area (Nl*Nm)

// Define Source and Output positions
#define Sl 100
#define Sm 80
#define Sp 70

#define Rl 100
#define Rm 140
#define Rp 70

int main()
{

    // ------------------------------------------------
    // Simulation parameters
    ReaL SR         = 44100.0;              // Sample Rate
    int NF          = 44100;                // Duration
    ReaL c          = 344.0;
    ReaL k          = 1/SR;
    ReaL h          = sqrt(3.0)*c*k;
    ReaL l2         = 1.0/3.0;

    //------------------------------------------------
    // Initialise input
    size_t pr_size  = sizeof(ReaL);
    ReaL *si        = (ReaL *)calloc(NF,pr_size);
    int dur = 20;
    int n;
    for(n=0;n<dur;n++){
        si[n] = 0.5*(1.0-cos(2.0*pi*n/(ReaL)dur));
    }

    size_t mem_size = area*Np;
    ReaL *dummy_ptr;
    ReaL ins;
    int cp,L,M,P;
```

```
    //-------------------------------------------
    // initialise memory
    ReaL *out  = (ReaL *)calloc(NF,pr_size);
    ReaL *u    = (ReaL *)calloc(mem_size,sizeof(ReaL));
    ReaL *u1   = (ReaL *)calloc(mem_size,sizeof(ReaL));
    if (out==NULL || u==NULL || u1==NULL) {
        printf("\nMemory alloc Failed\n");
    }

    time_t start = time(NULL);

    for(n=0;n<NF;n++)
    {
        // update u matrix
        for(P=1;P<Np-1;P++){
            for(M=1;M<Nm-1;M++){
                for(L=1;L<Nl-1;L++){

                    cp    = P*area+(M*Nl+L);
                    u[cp] = l2*(u1[cp-1]  + u1[cp+1]    + u1[cp-Nl]
                               +u1[cp+Nl] + u1[cp-area] + u1[cp+area]) - u[cp];

                }
            }
        }

        // sum in source
        u[(Sp*area)+(Sm*Nl+Sl)] += si[n];

        // read output
        out[n]  = u[(Rp*area)+(Rm*Nl+Rl)];

        // update pointers
        dummy_ptr = u1;
        u1 = u;
        u  = dummy_ptr;
    }

    time_t end = time(NULL);
    printf("\nProcess_time_:_%ld_seconds\n", (end-start) );

    // print last samples, and write output file
    printLastSamples(out, NF, 5);

    //-------------------------------------------
    free(si); free(out); free(u); free(u1);
    printf("\nSimuation_complete...\n\n");

    return 0;
}
```

B.2 POSIX thread C code for test simulation

```
#define ReaL double
#include "CJW_Audio.h"
#include <pthread.h>

#define NUM_THREADS 8

// Define Grid dims
#define Nl 256
#define Nm 296
#define Np 208

#define area (Nl*Nm)

// Define Source and Read
#define Sl 100
#define Sm 80
#define Sp 70

#define Rl 100
#define Rm 140
#define Rp 70

struct thread_data{
    int  thread_id;
    ReaL *u;
    ReaL *u1;
    ReaL l2;
};


void *updateScheme(void *targ){

    struct thread_data *md;

    md = (struct thread_data *) targ;

    int cp,L,M,P;
    int psize = Np/NUM_THREADS;

    int td = md->thread_id;
    int ps = psize*td;
    int pe = ps+psize;

    for(P=ps;P<pe;P++){

        if (P>0 && P<Np-1) {
            for(M=1;M<Nm-1;M++){
                for(L=1;L<Nl-1;L++){

                    cp    = P*area+(M*Nl+L);
                    md->u[cp] = md->l2*(md->u1[cp-1]+md->u1[cp+1]+md->u1[cp-Nl]+md->
                        u1[cp+Nl]+md->u1[cp-area]+md->u1[cp+area]) - md->u[cp];

                }
            }
        }
    }
    return NULL;
}

int main()
{

    // ------------------------------------------
    // Simulation parameters
```

```
    ReaL SR         = 44100.0;             // Sample Rate
    int NF          = 44100;
    ReaL c          = 344.0;
    ReaL k          = 1/SR;
    ReaL h          = sqrt(3.0)*c*k;
    ReaL l2         = 1.0/3.0;

    //-----------------------------------------
    // Initialise input
    size_t pr_size  = sizeof(ReaL);
    ReaL *si        = (ReaL *)calloc(NF,pr_size);
    int dur = 20;
    int n;
    for(n=0;n<dur;n++){
        si[n] = 0.5*(1.0-cos(2.0*pi*n/(ReaL)dur));
    }

    size_t mem_size = area*Np;
    ReaL *dummy_ptr;

    //-----------------------------------------
    // initialise memory
    ReaL *out  = (ReaL *)calloc(NF,pr_size);
    ReaL *u    = (ReaL *)calloc(mem_size,sizeof(ReaL));
    ReaL *u1   = (ReaL *)calloc(mem_size,sizeof(ReaL));
    if (out==NULL || u==NULL || u1==NULL) {
        printf("\nMemory alloc Failed\n");
    }

    pthread_t threads[NUM_THREADS];
    struct thread_data td[NUM_THREADS];
    int i;

    time_t start = time(NULL);

    for(n=0;n<NF;n++)
    {
        // update u matrix
        for (i=0; i<NUM_THREADS; i++) {
            td[i].thread_id = i;
            td[i].u = u;
            td[i].u1 = u1;
            td[i].l2 = l2;
            pthread_create(&threads[i], NULL, updateScheme, (void *) &td[i]);

        }

        for (i=0; i<NUM_THREADS; i++) {
            pthread_join(threads[i], NULL);
        }

        // sum in source
        u[(Sp*area)+(Sm*Nl+Sl)] += si[n];

        // non-interp read out
        out[n]  = u[(Rp*area)+(Rm*Nl+Rl)];

        // update pointers
        dummy_ptr = u1; u1 = u; u  = dummy_ptr;
    }

    time_t end = time(NULL);
    printf("\nProcess time : %ld seconds\n", (end-start) );

    // print last samples, and write output file
    printLastSamples(out, NF, 5);

    free(si); free(out); free(u); free(u1);
    return 0;
}
```

B.3 CUDA code for test simulation using 3D tiling

```
#define ReaL double
#include "CJW_Audio.h"
#include "CJW_Cuda.h"

// Define Grid dims, modulo Thread block
#define Nl 256
#define Nm 296
#define Np 208

#define area (Nl*Np)

// Define Thread block size
#define Bl 32
#define Bm 4
#define Bp 2

// Define Source and Read
#define Sl 100
#define Sm 80
#define Sp 70

#define Rl 100
#define Rm 140
#define Rp 70

// kernel methods
__global__ void UpDateScheme(ReaL *u, const ReaL * __restrict__ u1, ReaL l2);
__global__ void inout(ReaL *u,ReaL *out,ReaL ins,int n);

// ------------------------------------------------

int main(){

    // Set device number
    cudaSetDevice(2);
    printDevDetails();

    // ------------------------------------------------
    // Simulation parameters
    ReaL SR         = 44100.0;
    int NF          = 4410;
    ReaL c          = 344.0;
    ReaL k          = 1/SR;
    ReaL h          = sqrt(3.0)*c*k;
    ReaL l2         = 1.0/3.0;

    //------------------------------------------------
    // Initialise input
    int n;
    size_t pr_size  = sizeof(ReaL);
    int dur         = 20;
    ReaL *si_h      = (ReaL *)calloc(NF,pr_size);
    for(n=0;n<dur;n++){
        si_h[n] = 0.5*(1.0-cos(2.0*pi*n/(ReaL)dur));
    }

    // ------------------------------------------------
    // Set up grid and blocks
    int Gl = Nl/Bl;
    int Gm = Nm/Bm;
    int Gp = Np/Bp;

    dim3 dimBlockInt(Bl, Bm, Bp);
    dim3 dimGridInt(Gl, Gm, Gp);
```

```
    dim3 dimBlockIO(1, 1, 1);
    dim3 dimGridIO(1, 1, 1);

    size_t mem_size = area*Np*pr_size;
    ReaL *out_d, *u_d, *u1_d, *dummy_ptr;
    ReaL ins;

    //---------------------------------------------
    // Initialise memory on device
    cuErr( cudaMalloc(&u_d, mem_size));     cuErr( cudaMemset(u_d, 0, mem_size));
    cuErr( cudaMalloc(&u1_d, mem_size));    cuErr( cudaMemset(u1_d, 0, mem_size));
    cuErr( cudaMalloc(&out_d, NF*pr_size)); cuErr( cudaMemset(out_d, 0, NF*pr_size));

    //---------------------------------------------
    // initialise memory on host
    ReaL *out_h  = (ReaL *)calloc(NF,pr_size);
    if((out_h == NULL)){
        printf("\nout_h memory alloc failed...\n");
        exit(EXIT_FAILURE);
    }

    //---------------------------------------------
    // Compute scheme
    //---------------------------------------------

    cuErr( cudaDeviceSynchronize() );
    time_t start = time(NULL);

    for(n=0;n<NF;n++)
    {

        UpDateScheme<<<dimGridInt,dimBlockInt>>>(u_d,u1_d,l2);

        // perform read in out
        ins = si_h[n];
        inout<<<dimGridIO,dimBlockIO>>>(u_d,out_d,ins,n);

        // update pointers
        dummy_ptr = u1_d;
        u1_d = u_d;
        u_d = dummy_ptr;

    }

    // print process time
    checkLastCUDAError("Kernel");
    cuErr( cudaDeviceSynchronize() );
    time_t end = time(NULL);
    printf("\nProcess time : %ld seconds\n", (end-start) );

    // copy result back from device
    cuErr( cudaMemcpy(out_h, out_d, NF*pr_size, cudaMemcpyDeviceToHost) );

    // print last samples, and write output file
    printLastSamples(out_h, NF, 5);

    //---------------------------------------------
    // Free memory
    free(si_h);free(out_h);
    cudaFree(out_d);cudaFree(u_d);cudaFree(u1_d);

    exit(EXIT_SUCCESS);
}
```

```
//------------------------------------------------------
// Standard 3D update scheme
__global__ void UpDateScheme(ReaL *u, const ReaL * __restrict__ u1, ReaL l2)
{

    // get L,M,P from thread and block Id's
    int L = blockIdx.x * Bl + threadIdx.x;
    int M = blockIdx.y * Bm + threadIdx.y;
    int P = blockIdx.z * Bp + threadIdx.z;

    // Test that not at boundary
    if( (L>0) && (L<(Nl-1)) && (M>0) && (M<(Nm-1)) && (P>0) && (P<(Np-1)) ){

        // get linear position
        int cp = P*area+(M*Nl+L);

        u[cp] = l2*(u1[cp-1]+u1[cp+1]+u1[cp-Nl]+u1[cp+Nl]+u1[cp-area]+u1[cp+area]) -
            u[cp];

    }

}

//------------------------------------------------------
// read output and sum in input
__global__ void inout(ReaL *u,ReaL *out,ReaL ins,int n)
{
        // sum in source
        u[(Sp*area)+(Sm*Nl+Sl)] += ins;

        // non-interp read out
        out[n]  = u[(Rp*area)+(Rm*Nl+Rl)];

}
```

B.4 3D tiling with shared memory CUDA code kernel

```
__global__ void UpDate(double *u,double *u1,double L2)
{
   __shared__ double uS1[Bl][Bm];

   int tdl = threadIdx.x;
   int tdm = threadIdx.y;

   int L = blockIdx.x * Bl + tdx;
   int M = blockIdx.y * Bm + tdy;
   int P = blockIdx.z;

   int cp = P*area+(M*Nl+L);
   real sum = 0.0;

   // Load shared
   uS1[tdl][tdm] = u1[cp];
   __syncthreads();

   // Test that not at boundary
   if( (L>0) && (L<(Nl-1)) && (M>0) && (M<(Nm-1)) && (P>0) && (P<(Np-1)) ){

      if(tdl==0) sum+= u1[cp-1];
      else sum+= uS1[tdl-1][tdm];

      if(tdl==Bl-1) sum+= u1[cp+1];
      else sum+= uS1[tdl+1][tdm];

      if(tdm==0) sum+= u1[cp-Nl];
      else sum+= uS1[tdl][tdm-1];

      if(tdm==Bm-1) sum+= u1[cp+Nl];
      else sum+= uS1[tdl][tdm+1];

      u[cp] = L2*(sum+u1[cp-area]+u1[cp+area]) - u[cp];
   }
}
```

B.5 3D tiling with extended shared memory CUDA code kernel

```
__global__ void UpDate(double *u,double *u1,double L2)
{
   __shared__ double uS1[Bl+2][Bm+2];

   int tdl = threadIdx.x;
   int tdm = threadIdx.y;

   int L = blockIdx.x * Bl + tdx;
   int M = blockIdx.y * Bm + tdy;
   int P = blockIdx.z;

   // get linear position
   int cp   = P*area+(M*Nl+L);

   // Load shared
   tdl++; tdm++;
   uS1[tdl][tdm] = u1[cp];

   if ( (tdm==1) && !(M==0) ){
      uS1[tdl][tdm-1] = u1[cp-Nl];
   }
   if ( (tdm==BmS) && !(M==(Nm-1)) ){
      uS1[tdl][tdm+1] = u1[cp+Nl];
   }
   if ( (tdl==1) && !(L==0) ){
      uS1[tdl-1][tdm] = u1[cp-1];
   }
   if ( (tdl==BlS) && !(L==(Nl-1)) ){
      uS1[tdl+1][tdm] = u1[cp+1];
   }
   __syncthreads();

   // Test that not at boundary halo
   if( (L>0) && (L<(Nl-1)) && (M>0) && (M<(Nm-1)) && (P>0) && (P<(Np-1)) ){

      u[cp] = L2*(uS1[tdl-1][tdm]+uS1[tdl+1][tdm]
                  +uS1[tdl][tdm-1]+uS1[tdl][tdm+1]
                  +u1[cp-area]+u1[cp+area]) - u[cp];
   }
}
```

B.6 2D slicing with shared memory CUDA code kernel

```
__global__ void UpDate(double *u,double *u1,double L2)
{
   __shared__ double uS1[Bl][Bm];

   int tdl = threadIdx.x;
   int tdm = threadIdx.y;

   // Get 3D position
   int L = blockIdx.x * Bl + tdl;
   int M = blockIdx.y * Bm + tdm;
   int P,cp;

   // Initial variables
   double u1cpm = 0.0;
   double u1cp  = u1[area+(M*Nl+L)];
   double u1cpp,sum;

   for(P=1;P<(Np-1);P++){

      // Get linear position
      cp = P*area+(M*Nl+L);
      u1cpp  = u1[cp+area];
      // load shared
      uS1[tdl][tdm] = u1cp;
      __syncthreads();

      if( (L>0) && (L<(Nl-1)) && (M>0) && (M<(Nm-1)) ){

        sum = 0.0;
        if(tdl==0) sum+= u1[cp-1];
        else sum+= uS1[tdl-1][tdm];

        if(tdl==BlL-1) sum+= u1[cp+1];
        else sum+= uS1[tdl+1][tdm];

        if(tdm==0) sum+= u1[cp-Nl];
        else sum+= uS1[tdl][tdm-1];

        if(tdm==BmL-1) sum+= u1[cp+Nl];
        else sum+= uS1[tdl][tdm+1];

        u[cp] = L2*(sum+u1cpm+u1cpp) - u[cp];

        u1cpm = u1cp;
        u1cp  = u1cpp;
        __syncthreads();
      }
   }
}
```

B.7 2D slicing with extended shared memory CUDA code kernel

```
__global__ void UpDate(double *u,double *u1,double L2)
{
   __shared__ double uS1[Bl+2][Bm+2];

   int tdl = threadIdx.x;
   int tdm = threadIdx.y;

   int L = blockIdx.x * Bl + tdl;
   int M = blockIdx.y * Bm + tdm;

   int P,cp;
   double u1cpm = 0.0;
   double u1cp  = u1[area+(M*Nl+L)];
   double u1cpp;
   tdl++; tdm++;

   for(P=1;P<(Np-1);P++){

      cp = P*area+(M*Nl+L);
      u1cpp  = u1[cp+area];
      // load shared
      uS1[tdl][tdm] = u1cp;

      if ( (tdm==1) && !(M==0) ){
        uS1[tdl][tdm-1] = u1[cp-Nl];
      }
      if ( (tdm==BmL) && !(M==(Nm-1)) ){
        uS1[tdl][tdm+1] = u1[cp+Nl];
      }
      if ( (tdl==1) && !(L==0) ){
        uS1[tdl-1][tdm] = u1[cp-1];
      }
      if ( (tdl==BlL) && !(L==(Nl-1)) ){
        uS1[tdl+1][tdm] = u1[cp+1];
      }
      __syncthreads();

      if( (L>0) && (L<(Nl-1)) && (M>0) && (M<(Nm-1)) ){

        u[cp] = L2*(uS1[tdl-1][tdm]+uS1[tdl+1][tdm]
                   +uS1[tdl][tdm-1]+uS1[tdl][tdm+1]
                   +u1cpm+u1cpp) - u[cp];

        u1cpm = u1cp;
        u1cp  = u1cpp;
        __syncthreads();
      }
   }
}
```

B.8 Time loop for asynchronous four GPU device implementation

```
for(n=0;n<NF;n++)
{
   // Compute halo layers, then interior
   hp = 0;
   for(i=0;i<num_gpus;i++){
        cudaSetDevice(gpu[i]);
        UpDateHalo<<<dimGridHalo,dimBlockHalo,0,stream_halo[i]>>>(u_d[i],u1_d[i],hpos
            [hp]);
        hp++;
        if(i>0 && i<num_gpus-1){
                UpDateHalo<<<dimGridHalo,dimBlockHalo,0,stream_halo[i]>>>(u_d[i],u1_d
                    [i],hpos[hp]);
                hp++;
        }
        cudaStreamQuery(stream_halo[i]);
        UpDateInterior<<<dimGridInt,dimBlockInt,0,stream_int[i]>>>(u_d[i],u1_d[i],i);
   }
   for(i=0;i<num_gpus;i++){
      cudaSetDevice(gpu[i]);
      cudaDeviceSynchronize();
   }

   // Exchange Halos
  cudaMemcpyPeerAsync(u_d[1],gpu[1],&u_d[0][pos[0]],gpu[0],area_size,stream_halo[0]);
  cudaMemcpyPeerAsync(u_d[2],gpu[2],&u_d[1][pos[1]],gpu[1],area_size,stream_halo[1]);
  cudaMemcpyPeerAsync(u_d[3],gpu[3],&u_d[2][pos[1]],gpu[2],area_size,stream_halo[2]);
  for(i=0;i<num_gpus;i++){
     cudaStreamSynchronize(stream_halo[i]);
  }
  cudaMemcpyPeerAsync(&u_d[0][pos[1]],gpu[0],&u_d[1][area],gpu[1],area_size,
      stream_halo[1]);
  cudaMemcpyPeerAsync(&u_d[1][pos[2]],gpu[1],&u_d[2][area],gpu[2],area_size,
      stream_halo[2]);
  cudaMemcpyPeerAsync(&u_d[2][pos[2]],gpu[2],&u_d[3][area],gpu[3],area_size,
      stream_halo[3]);

   // perform I/O
   ins = 0.0;
   if(n<alength) ins = sib_h[n];
   cudaSetDevice(gpu[Scard]);
   UpdateInput<<<dimGridIO,dimBlockIO>>>(u_d[Scard], ins, Sindex);
   cudaSetDevice(gpu[Rcard]);
   UpdateOutput<<<dimGridIO,dimBlockIO>>>(u_d[Rcard], out_d, n, Rindex);

   // Synchronise
   for(i=0;i<num_gpus;i++){
      cudaSetDevice(gpu[i]);
      cudaDeviceSynchronize();
      // update pointers
      dummy_ptr = u1_d[i];
      u1_d[i] = u_d[i];
      u_d[i] = dummy_ptr;
   }
}
```

B.9 FCC simulation CUDA code kernel

```
__global__ void UpDateScheme(ReaL *u, const ReaL * __restrict__ u1)
{

   // get X,Y,Z from thread and block Id's
   int X = blockIdx.x * Bx + threadIdx.x;
   int Y = blockIdx.y * By + threadIdx.y;
   int Z = blockIdx.z * Bz + threadIdx.z;

   // Test that not at halo
   if( (X>1) && (X<(Nx-2)) && (Y>1) && (Y<(Ny-2)) && (Z>1) && (Z<(Nz-2)) ){

      // get linear position
      int cp = Z*area+(Y*Nx+X);

      // Load differing sum parts
      ReaL s1 = u1[cp+1+Nx];
      ReaL s2 = u1[cp-1+Nx];
      ReaL s3 = u1[cp+Nx-area];
      ReaL s4 = u1[cp+Nx+area];

      ReaL s5 = u1[cp+1-Nx];
      ReaL s6 = u1[cp-1-Nx];
      ReaL s7 = u1[cp-Nx-area];
      ReaL s8 = u1[cp-Nx+area];

      ReaL ps;

      if ((X+Z)%2==0){
         ps = s1 + s2 + s3 + s4;
      }
      else{
         ps = s5 + s6 + s7 + s8;
      }

      u[cp] = l2*0.5*(u1[cp+1]     +u1[cp-1]     +u1[cp-1-area]+u1[cp+1-area]
                     +u1[cp-1+area]+u1[cp+1+area]+u1[cp-area]  +u1[cp+area] + ps
                      - 12.0*u1[cp]) + 2.0*u1[cp] - u[cp];

      }
}
```

B.10 Frequency-independent lossy boundary CUDA code kernel

```
__global__ void UpDateScheme(ReaL *u, const ReaL * __restrict__ u1, ReaL l2, ReaL fac
    , ReaL fac2)
{

   // get X,Y,Z from thread and block Id's
   int X = blockIdx.x * Bx + threadIdx.x;
   int Y = blockIdx.y * By + threadIdx.y;
   int Z = blockIdx.z * Bz + threadIdx.z;

   // Test that not at halo
   if( (X>0) && (X<(Nx-1)) && (Y>0) && (Y<(Ny-1)) && (Z>0) && (Z<(Nz-1)) ){

      // get linear position
      int cp    = Z*area+(Y*Nx+X);

      // local variables
      ReaL fcc  = 1.0;
      ReaL fcc2 = 1.0;

      int K    = (0||(X-1)) + (0||(X-(Nx-2))) + (0||(Y-1)) + (0||(Y-(Ny-2)))
               + (0||(Z-1)) + (0||(Z-(Nz-2)));

      // set loss coeffs at walls
      if(K<6){
         fcc  = fac;
         fcc2 = fac2;
      }

      // Get sum of neighbours
      ReaL S   = u1[cp-1]+u1[cp+1]+u1[cp-(Nx+2)]+u1[cp+(Nx+2)]
                +u1[cp-area]+u1[cp+area];

      // Calc update
      u[cp]    = fcc*((2.0-K*l2)*u1[cp] + l2*S - fcc2*u[cp]);

      }

}
```

B.11 Optimised Engquist Majda boundary CUDA code kernel

```
__global__ void UpDateInterior(ReaL *u,const ReaL * __restrict__ u1,ReaL *u2,ReaL *
    u1L,ReaL *u1R)
{
   // get coords from thread and block Id's
   int X = blockIdx.x * Bx + threadIdx.x;
   int Y = blockIdx.y * By + threadIdx.y;
   int Z = blockIdx.z * Bz + threadIdx.z;

   int cp    = (Z*area)+(Y*DIM+X);
   ReaL u1cp = u1[cp];

   // Load up u1 Left and Right 2D arrays, Z major so coalesced in Face update
   if(X==0){
      u1L[Y*DIM+Z] = u1cp;
      u1L[area+(Y*DIM+Z)] = u1[cp+1];
   }
   if(X==DIM-1){
      u1R[Y*DIM+Z] = u1cp;
      u1R[area+(Y*DIM+Z)] = u1[cp-1];
   }

   // Test that not at faces
   if( (X>0) && (X<DIM-1) && (Y>0) && (Y<DIM-1) && (Z>0) && (Z<DIM-1) ){

      ReaL usum = u1[cp-1]+u1[cp+1]+u1[cp-DIM]+u1[cp+DIM]+u1[cp-area]+u1[cp+area];

      // Pickup coefficients a0 and a1 from constant memory.
      u[cp] = -u2[cp] + cf_d[0].a0*u1cp + cf_d[0].a1*usum;

   }

}
```